

7

Numerical Analysis

Numerical analysis deals with the computational issues of applied mathematics. The focus of this field is on “results-oriented” mathematics. That an expression can be represented as an infinite recursion or sum is only the beginning for numerical analysis. Numerical mathematics must also deal with practical issues, like the number of terms that are needed for a given accuracy, and the speed compared to alternative techniques.

It is often necessary to estimate the values of data sets in between samples. The data points may not be equally spaced and may not be represented by a convenient function. The goal of **interpolation** is to produce a smooth curve that passes through all the data points.

Finite sums and differences are the tools that the numerical mathematician needs to perform integration and differentiation. Numerical calculus is valuable because it facilitates the evaluation of even the most difficult mathematical expressions. Numerical **integration** and **differentiation** can be performed on tables of data as well as analytical equations.

This chapter deals with the following numerical techniques:

1. Interpolation
2. Integration
3. Differentiation
4. Function minimization

7.1 INTERPOLATION

Interpolation is something of an art. In practice, it is often impossible to objectively evaluate the performance of any interpolation approach, since it depends on the true (and, in general, unknown) values of the data in between the known points. Interpolation performance is often judged subjectively. The question is asked: Does the interpolating function produce a curve that is pleasing to the eye? The answer is usually **yes**, as long as the data is oversampled and not corrupted with noise.

7.1.1 Piecewise Interpolation

For any arbitrary set of $n + 1$ data points, an n th order polynomial can always be found that passes through all of the points. For even medium size data sets, numerical precision can limit the polynomial order that can be chosen. Over 100 years ago, Runge noted that the problems of high order polynomial interpolation are compounded by large oscillations in between the data points.

One solution is to divide the data interval into pieces and interpolate each piece independently. For polynomial interpolation of sparsely sampled data, this tends to produce sharp cusps at the edges of each subinterval. Spline approaches eliminate these cusps by matching the slope at the edges of each segment.

7.1.2 Linear Interpolation

Linear interpolation is the most straightforward interpolation method. Data points are simply connected with straight line segments. Each interpolated value lies along an appropriate straight line segment. For any two points, the interpolation at x is given by

$$y = [y_1(x - x_2) - y_2(x - x_1)] / (x_1 - x_2)$$

where (x_1, y_1) and (x_2, y_2) are the known data points.

Linear interpolation is a special case of the Lagrange formula described in the following section.

7.1.3 Lagrange n -Point Interpolation

Lagrange interpolation is an extension of the linear interpolation formula. This is one of the most versatile interpolation approaches. At each data point, this technique computes the $(n - 1)$ th order polynomial that passes through the ordinates of each group of n neighboring points. If the data are samples from an m th order polynomial, and $m < n$, the interpolation is exact everywhere. The interpolation is always exact at the input data samples (many interpolation techniques do not have this property).

The one problem with Lagrange interpolation is that the derivative of the polynomial is totally unconstrained. If the data points are far apart, this causes the interpolation function to have sharp cusps at many points. Also, for large orders, the polynomial functions have oscillations in between the data points.

See Functions: `interp()`, `interp1()`.

7.1.4 Cubic Spline Interpolation

Some of the drawbacks of polynomial interpolation are overcome with spline functions. A spline is a low order interpolating polynomial with slope and curvature con-

straints. The cubic spline interpolates between data points with the following third order equation:

$$f_i(x) = b_{0i} + b_{1i}x + b_{2i}x^2 + b_{3i}x^3$$

where $x_i \leq x \leq x_{i+1}$.

The smooth appearance of the spline results from matching the slope and the curvature of the cubic formulas in adjacent intervals. Noting that the second derivative of a cubic is a line, we can develop a tridiagonal system of equations to constrain the slope and curvature of the cubic formulas across adjacent intervals of the data. All that is needed to solve this set of equations is the specification of the second derivative at the endpoints of the data. Often it is convenient to assume that the second derivative at the endpoints of the data is zero. This condition corresponds to a natural spline interpolant.

Interpolating with a cubic spline is similar to interpolating with a second order Lagrange polynomial. This is because the slope constraints of the spline are analogous to an order constraint for the Lagrange polynomial. The major difference is that, in general, the second order Lagrange polynomial will not be smooth at all the data points. However, cubic spline interpolation is a little more difficult to use than polynomial interpolation since the second derivatives must be computed over the entire interval of interest (and specified at the endpoints).

See Function: spline().

7.1.5 Coordinates of a Centroid

It frequently happens that a variable or function is given in tabular form and that the abscissa values are of more interest than the ordinates:

1. In spectral analysis, one may wish to determine the frequency of an inharmonic sinusoid.
2. In image processing, the position of a light source that straddles several camera pixels may be of interest (e. g., a pixel tracker).
3. In one-dimensional correlation procedures, estimates of fractional time lags are often required to determine the pure delay between two similar time series.
4. In two-dimensional correlation procedures, determining the amount of spatial misregistration between two similar images may be important.

For these cases, the **coordinates of the centroid** are more informative than the actual value of the centroid. Centroiding techniques are ideal for these applications. Because of the focus on the coordinate values, centroiding is sometimes referred to as **inverse interpolation**.

For sampled data, the one-dimensional equation for the coordinates of a centroid is

$$c_x = \frac{\sum_i i f(i)}{\sum_i f(i)}$$

This equation is simple and theoretically correct given infinite digital resolution and infinite signal-to-noise ratio. However, for real data with low signal-to-noise ratios, straightforward application of this formula can lead to more than half a resolution bin of error. **This amount of error defeats the entire purpose of the centroid.**

The reader can easily verify the inaccuracy of this formula by applying it to the frequency spectrum of a low amplitude sinusoid plus white noise. The sinusoid can be set to a known (inharmonic) frequency. The coordinates of the centroid will appear to fluctuate, depending on the number of bins included in the computation of the centroid. This is because the inclusion of noise bins biases the estimate of the centroid as much as half a resolution bin.

One solution is to decide which bins are “valid” (signal) and to exclude the “invalid” (noise) bins from the centroid calculation. If the signal or function of interest is known to be **unimodal**, then a simple threshold can be set. The assumption is that the data is a quantized version of a function that has a single maximum and is monotonically decreasing below this value.

Let u_f be the mean of the absolute value of the data computed over all the bins of the centroid. Assuming an equally likely distribution of the noise (i. e., white noise) throughout the bins, a simple decision strategy results:

```
bin  $i$  is "valid" if  $|f(i)| \geq u_f$ 
bin  $i$  is "invalid" otherwise
```

The centroid is then computed over all the “valid” bins. For low signal-to-noise ratios, this strategy can lead to an order of magnitude improvement in the estimate of the centroid coordinates compared to the original quantization resolution.

This centroiding strategy is easily extended to two dimensions and yields comparable results.

See Functions: mxcentro(), v.centro().

7.2 INTEGRATION

Numerical integration provides quick and accurate solutions to difficult integrals or integrals without analytical solutions. Specifically, these approaches are used to estimate definite integrals of the following form:

$$y(x) = \int_{x_0}^{x_1} f(x) dx$$

where x_0 and x_1 are the endpoints of the interval of interest and $f(x)$ is any one-dimensional function of x .

Numerical integration techniques involve fitting simple curves to data over subintervals of the domain of interest. The simple curves are easily integrated and can be summed together to approximate the total integral.

7.2.1 Rectangular Rule

The simplest integration formula is the Rectangular Rule. This rule divides the integration interval into rectangular panels. Each panel has a height equal to the value

of the function at a data point and a width equal to the distance between data points. If the distance between data points is small, then the approximation to the integral is good. For data that is equally spaced, the integral I is

$$I = dx(y_0 + y_1 + y_2 + \dots + y_{n-1})$$

where dx is the distance between data points and y_i is the value of the function at x_i .

7.2.2 Trapezoidal Rule

The Trapezoidal Rule connects data points with a straight line. The area of each panel is the average height times the panel width. For equally spaced data, the integral I is

$$I = dx[(y_0 + y_1)/2 + (y_1 + y_2)/2 + \dots + (y_{n-2} + y_{n-1})/2]$$

where dx is the distance between data points and y_i is the value of the function at x_i .

7.2.3 Romberg Integration

Romberg integration successively applies the Trapezoidal Rule to efficiently estimate the integral of a function. To demonstrate the approach, the first two integral approximations are

$$I_0 = 2dx[(y_0 + y_2)/2 + (y_3 + y_5)/2 + \dots + (y_{n-3} + y_{n-1})/2]$$

$$I_1 = dx[(y_0 + y_1)/2 + (y_1 + y_2)/2 + \dots + (y_{n-2} + y_{n-1})/2]$$

Using Richardson's improvement formula (see the Differentiation section 7.3.4), a more accurate estimate of the integral can be obtained by combining the I_0 and I_1 :

$$I = I_1 + (I_1 - I_0)/3$$

If the difference between I_0 and I_1 is small, then the approximation to the integral is good. Otherwise, the interval between data points is halved and the procedure is repeated.

One disadvantage of Romberg's approach is in the integration of periodic functions over an integer number of periods. If the grid locations of the integration panels nearly align with the zero crossings of the function, the Romberg approach may terminate prematurely with an incorrect answer. These situations can easily be avoided by segmenting the integration interval into lengths that are not integer multiples of the period.

See Function: romberg().

7.2.4 Simpson's Rule

Simpson's integration technique uses parabolic arcs to approximate the data. The data interval must be divided into an even number of equal panels of width dx . The most common form is the **one-third** rule:

$$I = dx[(y_0 + 4y_1 + 2y_2 + 4y_3 + \dots + 2y_{n-3} + 4y_{n-2} + y_{n-1})]/3$$

It is interesting to note that if the input data is parabolic, then Simpson's formula provides the exact integral.

See Function: `integrat()`.

7.3 DIFFERENTIATION

Difference formulas are used to approximate the derivatives of tabulated data as well as difficult analytical expressions. Although a multiplicity of formulas is available, for a given application some formulas are better than others. Central difference formulas are the most accurate. The forward and backward differences are also useful in evaluating the derivatives at the edges of tables of data.

7.3.1 First Order Differences

The first derivative of the function $f(x)$ can be approximated by

$$f'(x) = [f(x + dx) - f(x)]/dx$$

if dx is small. The truncation error of this approximation is proportional to the size of the increment dx (i. e., a first order approximation, $O(dx)$). If this equation is used as an estimate of the derivative at x , then it is called a **backward difference**. However, if it is used as an estimate of the derivative at $x + dx$, then it is called a **forward difference**. Forward differences estimate the derivative at a point with data that precedes the point. Similarly, backward differences approximate the derivative with data that follow the point.

A more accurate approximation to the derivative is given by the following equation:

$$f'(x) = [f(x + dx) - f(x - dx)]/(2dx)$$

where again it is assumed that dx is small. The truncation error of this approximation is proportional to dx^2 (i. e., a second order approximation, $O(dx^2)$). This two-sided equation is the basis of the **central difference** formulas. Central differences use an equal number of data points before and after the point of interest.

7.3.2 Higher Order Differences

Difference formulas allow the approximation of the n th derivative of a function. For tabulated data, it is assumed that the function passes through each discrete point. **In computing the derivative of tables of data, it is necessary that the data points be equally spaced.** The difference approximations consist of weighted local averages of the data.

The difference coefficients are related to Pascal's triangle and the binomial coefficients, as shown in the following table. Note that the alternating sign of these coefficients also depends on the order of the derivative. In the forward and backward difference formulas, the i th coefficient for the n th derivative is given by

$$C_{ni} = (-1)^{n+i} n!/[i!(n-i)!]$$

where $i = 0, 1, \dots, n$.

The coefficients of the difference formulas are the same for the forward and backward equations. The only real distinction between these formulas is in their use, not in the coefficients themselves. In fact, for even order derivatives, the central difference coefficients are the same as the forward and backward coefficients. For odd order derivatives, the central difference coefficients are not the same as the forward and backward coefficients. For these cases, the middle coefficient is always zero.

nth Derivative n	Forward and Backward Difference Coefficients	Central Difference Coefficients
1	-1 1	-1 0 1
2	1 -2 1	1 -2 1
3	-1 3 -3 1	-1 2 0 -2 1
4	1 -4 6 -4 1	1 -4 6 -4 1
5	-1 5 -10 10 -5 1	-1 4 -6 0 6 -4 1

7.3.3 Examples of Difference Calculations

Before the difference (binomial) coefficients can be used to compute derivatives, they must be normalized by the sum of the absolute values of all the coefficients:

$$C_n = |C_{n0}| + |C_{n1}| + |C_{n2}| + \dots + |C_{nn}|$$

The coefficients are applied to the appropriate data points as a weighted local average about the point of interest.

For example, the second derivative at the point x_i is approximated by

$$\begin{aligned} y''(x_1) &= [C_{20}y(x_0) + C_{21}y(x_1) + C_{22}y(x_2)]/(C_2 dx) \\ &= [y(x_0) - 2y(x_1) + y(x_2)]/(4 dx) \end{aligned}$$

When the coefficients are applied in this way to successive data points, the process is called a **convolution**. Convolution is discussed in more detail in Chapter 8 on Digital Signal Processing.

Consider the following table of x, y data:

x	1.1	1.2	1.3	1.4	1.5	1.6	1.7
y	-2	4	-1	6	8	4	3

We will use this table of data for all the following examples to evaluate derivatives at specific points of interest.

Example 1:

Approximate the first derivative at $x = 1.4$.

We will use the central first difference formula normalized by $C_1 = 2$:

$$\begin{aligned} y'(1.4) &= [C_{10}y(1.3) + C_{11}y(1.5)]/(C_1 dx) \\ &= [-1 * y(1.3) + 1 * y(1.5)]/(2 * 0.1) \\ &= (1 + 8)/0.2 = 45 \end{aligned}$$

Example 2:

Approximate the first derivative at $x = 1.1$.

Since $x = 1.1$ is on the left edge of the table, we must use the backward first difference formula. The normalization factor is $C_1 = 2$.

$$\begin{aligned} y'(1.1) &= [C_{10}y(1.1) + C_{11}y(1.2)]/(C_1 dx) \\ &= [-1 * y(1.1) + 1 * y(1.2)]/(2 * 0.1) \\ &= (2 + 4)/0.2 = 30 \end{aligned}$$

Example 3:

Approximate the first derivative at $x = 1.7$.

Since $x = 1.7$ is on the right edge of the table, we must use the forward first difference formula. The normalization factor is $C_1 = 2$.

$$\begin{aligned} y'(1.7) &= [C_{10}y(1.6) + C_{11}y(1.7)]/(C_1 dx) \\ &= [-1 * y(1.6) + 1 * y(1.7)]/(2 * 0.1) \\ &= (-4 + 3)/0.2 = -5 \end{aligned}$$

7.3.4 Improving the Difference Estimates

For the lower order differences, there are several other formulas that can provide slightly better estimates of the derivatives. These formulas operate across more data points than those previously described. For m -points, the formulas represent the derivatives of an $m - 1$ order polynomial that passes through the points. If the polynomial matches the data well, then so does the derivative.

For example, a five-point central difference formula for estimating the first derivative is shown in the following table:

offset	-2	-1	0	1	2
coefficient	1/12	-2/3	0	2/3	-1/12

For data that is sufficiently oversampled, this five-point formula usually provides a better estimate of the first derivative than the three-point formula described previously.

Finally, it should be mentioned that all the central difference formulas can be modified to interpolate between samples as they differentiate. This is useful if the derivative is needed in between data points. The derivative estimates are best at the central data points.

Richardson's Improvement Theorem. Richardson's improvement theorem is of fundamental importance to numerical approximations with truncated series. This approach can be used to improve estimates of the derivatives of any differentiable function. Consider two estimates of the derivative, each made with a different step size. Essentially, Richardson's formula computes the relative weighting of each of these estimates needed to form a more accurate approximation of the derivative.

This theorem is as follows:

Suppose that an approximation is of a known truncation order, $O(h^n)$. Suppose also that this approximation is evaluated for two arbitrary step sizes, h and rh , where $r > 1$. If $F(h)$ is the evaluation for step size h , and $F(rh)$ is the evaluation for step size rh , then an improved estimate $F_1(h)$ may be formed according to:

$$F_1(h) = [r^n F(h) - F(rh)] / (r^n - 1)$$

This new estimate will have a smaller truncation error than either $F(h)$ or $F(rh)$. That is, the truncation order of $F_1(h)$ will be larger than n .

This improvement is sometimes called **Romberg extrapolation**, since it is also used in connection with the trapezoidal rule of integration. Since the trapezoidal rule is an $O(h^2)$ approximation, $n = 2$. The ratio of the step sizes r is 2, since Romberg's integration approach changes its step size by factors of two. Recall that the weighting on the integral approximation with the smaller step size was $4/3 = r^n / (r^n - 1)$, whereas the weighting on the integral approximation with the larger step size was $-1/3 = -1 / (r^n - 1)$.

See Functions: deriv(), deriv1().

7.4 FUNCTION MINIMIZATION

Even the simplest nonlinear equation can have no analytical solution. This section describes the numerical minimization of differentiable functions. There are several approaches, and the performance of each is highly dependent on the function that is minimized.

In curve fitting, the function to be minimized is often nonnegative, such as mean-squared error. For these applications, root-finding is equivalent to function minimization. The **Newton-Raphson** one-dimensional root-finding approach is described next.

For multidimensional smooth functions, gradient minimization techniques are very robust. The method of **conjugate gradients** is discussed in this section, since it is one of the more effective approaches.

7.4.1 Finding Roots of One-Dimensional Functions

One of the most common and difficult problems in mathematical analysis is the solution of nonlinear equations. For example, consider the following equation:

$$\sin(x) = 3 * x + 0.1$$

Although this is a very simple equation, there is no analytical method for solving for x . All attempts at a straightforward solution are of no avail. Numerical root-finding algorithms are ideal for problems such as these.

As indicated by the term “root-finding,” much of the early work in numerical minimization dealt with polynomials. However, most of these techniques work just as well with any differentiable function.

Newton–Raphson Method. The Newton–Raphson technique is a simple iterative approach for finding the value of x where $f(x) = 0$. The formula results from the first two terms of the Taylor series expansion at x_i :

$$f(x) = f(x_i) + f'(x_i) * (x - x_i) + \dots$$

Setting $f(x) = 0$ and solving for $x = x_{i+1}$ yields

$$x_{i+1} = x_i - f(x_i)/f'(x_i)$$

The value of the function is determined at x_{i+1} , and if it is not close enough to zero, the iteration is repeated.

One very nice feature of the Newton–Raphson technique is that the number of required iterations is usually insensitive to the initial guess. This is because the step size is weighted by the derivative of the function. The worse the initial guess, the larger the initial weighting. The number of iterations depends more on the behavior of $f'(x)$ as $f(x)$ approaches zero. For example, finding the zeros of $\sin(x)$ requires very few iterations, since the slope of the sine function is maximum at the zero-crossings. However, finding the zeros of $\cos(x) - 1$ requires comparatively more iterations since its derivative (and the weighted step size) vanishes at the zero-crossings. In either case, the number of required iterations is fairly independent of the initial guess.

In this form, the Newton–Raphson algorithm explicitly requires the derivative $f'(x)$. It is often more convenient to approximate the derivative with the two-point difference formula:

$$f'(x_i) = [f(x_i) - f(x_i + \delta)]/\delta$$

where a good choice for δ is

$$\delta = x_i * e^{-5}.$$

Incorporating this approximation, the new algorithm becomes

$$x_{i+1} = x_i - \delta * [-1 + f(x_i + \delta)/f(x_i)]^{-1}$$

This approximation alleviates the need to specify the derivative of the function, yet it maintains the convergence speed of the original Newton–Raphson approach.

Example:

Find the square root of 36. That is, find the zeros of

$$f(x) = 36 - x^2$$

For this case, the Newton–Raphson algorithm simplifies to

$$x_{i+1} = x_i + (36 - x_i^2)/(2x_i + \delta)$$

Choosing $\delta = 1.e - 5$ and a first guess of $x = 15$, the results of each iteration are as follows:

$$\begin{aligned}
 x_0 &= 15 \\
 x_1 &= 8.700003 \\
 x_2 &= 6.4189677 \\
 x_3 &= 6.0136734 \\
 x_4 &= 6.0000157 \\
 x_5 &= 6.0000000
 \end{aligned}$$

Thus, for this example, the Newton–Raphson algorithm converges to within eight significant digits of the correct answer in five steps.

See Function: `newton()`.

7.4.2 Finding Roots of Polynomials

The roots of polynomials have been of great interest to mathematicians over the centuries. The n th order polynomial of the form

$$f(x) = \text{coef}[0] + x * \text{coef}[1] + x^2 * \text{coef}[2] + \dots + x^{n-1} * \text{coef}[n - 1]$$

has, in general, n complex roots. Yet closed form solutions exist only for polynomials less than fifth order, and the solutions for third and fourth order polynomials are very cumbersome.

The Newton–Raphson method is a useful tool for the iterative evaluation of roots of differentiable functions (e. g., polynomials). Polynomials with real coefficients are analytic functions and are differentiable everywhere in the complex plane. When used in conjunction with the Cauchy–Riemann equations, the Newton–Raphson technique can be extended to evaluate complex roots of high order polynomials.

See Function: `p_roots()`.

7.4.3 Minimization of Multidimensional Functions

This section describes some numerical approaches to solving nonlinear equations that may have no analytical solution. The simplest multidimensional minimization techniques are the unconstrained gradient-search approaches. The negated gradient at a point on the surface points in the direction of greatest decrease. If the gradient exists, and if unimodality can be assumed, the minimum of the function can be found by successively following the surface in the direction of its steepest descent. For multimodal surfaces, multiple starting points are often required.

Gradient approaches work best when the scales of the independent variables are chosen so that the components of the typical gradient vector are of comparable magnitude. These search procedures can be very inefficient if a few components of the gradient are consistently much smaller than the rest.

Method of Steepest Descent. The method of steepest descent is most easily described in three dimensions. We wish to find the solution of the equation $f(x, y, z) = 0$. Let $f(x_0, y_0, z_0)$ be a first guess of the minimum. If this is not the minimum, the guess can be improved by considering the gradient of the function (f_x, f_y, f_z) at the guess:

$$\begin{aligned}x_1 &= x_0 - h * f_x(x_0, y_0, z_0) \\y_1 &= y_0 - h * f_y(x_0, y_0, z_0) \\z_1 &= z_0 - h * f_z(x_0, y_0, z_0)\end{aligned}$$

where h is a small value normalized by the magnitude of the gradient vector. The function is evaluated at (x_1, y_1, z_1) to verify that this estimate is closer to zero and the process is iterated until the desired accuracy is reached. If the new estimate is not closer to zero, the step size h is halved until a smaller error is found. If the surface is smooth and unimodal (e. g., quadratic), this procedure is guaranteed to find the minimum.

The Conjugate Gradient Approach. Although the method of steepest descent usually converges to the desired minimum, it is not the fastest gradient approach. This technique tends to zigzag down even the smoothest surface, since each new direction of descent tends to be orthogonal (but not conjugate) to the last direction. Thus, this algorithm tends to either overshoot or undershoot the surface fall line.

The **conjugate gradient** approach offers a solution to this problem. The n -dimensional approach searches along “conjugate directions,” which are weighted averages of the current gradient with the gradients of the last n steps. This averaging tends to smooth the search path, resulting in a much more rapid convergence than the method of steepest descent. For a more detailed description of the conjugate gradient approach, the reader should refer to the original work of Fletcher and Reeves (1964).

See Function: conjgrad().