Approximation 6

If native pow'r prevail not, shall I doubt
To seek for needful succor from without?

Vergil
(trans. John Dryden)

6.1 Motivation

6.1.1 Examples from Physics

Both in theoretical and in experimental physics we are often faced with a table of data points; we typically wish to manipulate physical quantities even for cases that lie "in between" the table rows. Here are a few examples:

1. **Velocity from unequally spaced points**
   When introducing numerical differentiation in chapter 3, our first motivational example came from classical mechanics. This, the most trivial example of differentiation in physics, takes the following form in one dimension:

$$v = \frac{dx}{dt} \tag{6.1}$$

   where v is the velocity and x is the position of the particle in a given reference frame. In that chapter, we spent quite a bit of time discussing how to compute finite differences, and how to estimate the error budget.

   The question then arises how we should handle the case where we know the positions at *unequally spaced points*. To clarify what we mean by that, imagine we only have access to a set of n discrete data points (i.e., a table) of the form (t_j, x_j) for $j = 0, 1, \ldots, n-1$; the positions are known only at fixed times t_j not of our choosing, which are *not* equally spaced. One could simply revert to the general finite-difference formulas (i.e., those not on a grid); since you don't control the placement of the points, you'd likely have to use a forward or backward formula. However, simply using a noncentral difference formula where the h is determined by whatever happened to be the distance between two t_j's is a recipe for disaster: what if there are many data points for early and late times, but few in between? Should we resign ourselves to not being able to extract the velocity at intermediate times?

An alternative approach is to take the table of unequally spaced input data points, produce an *interpolating polynomial* $p(t)$ which captures the behavior of the data effectively, and then evaluate the derivative of the interpolating polynomial *quasi-analytically*. As an aside, an analogous example arises when dealing with noisy data: one could first carry out, say, *Fourier smoothing* and then get a nicer looking derivative. This brings us to the next example, which also relates to Fourier analysis.

2. **Differentiating a wave function**
When taking an introductory quantum mechanics course, you may have wondered why we sometimes work in coordinate space and sometimes in momentum space. One of the reasons is that operations that may be difficult in one space are easy in another. A fascinating example of this is known as *Fourier differentiation* or *spectral differentiation*. To see what we mean, consider the following wave function expressed in terms of the inverse Fourier transform:

$$\psi(t,x) = \frac{1}{2\pi}\int_{-\infty}^{\infty} dk\, \tilde{\psi}(t,k)e^{ikx} \tag{6.2}$$

Taking the spatial derivative of $\psi(t,x)$ becomes a simple arithmetic operation (multiplication by a scalar) in momentum space:

$$\frac{\partial}{\partial x}\psi(t,x) = \frac{1}{2\pi}\int_{-\infty}^{\infty} dk\, \tilde{\psi}(t,k)\frac{\partial}{\partial x}e^{ikx} = \frac{1}{2\pi}\int_{-\infty}^{\infty} dk\, \tilde{\psi}(t,k)ike^{ikx} \tag{6.3}$$

In the first step we emphasized that only the complex exponential is position dependent and in the second step we took the derivative. As advertised, we replaced the derivative on the left-hand side with a simple multiplication on the right-hand side. We can take an extra step, expressing the $\tilde{\psi}(t,k)$ itself in terms of a (direct) Fourier transform:

$$\frac{\partial}{\partial x}\psi(t,x) = \frac{1}{2\pi}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} dk\, dy\, \psi(t,y)ike^{ik(x-y)} \tag{6.4}$$

Take a moment to appreciate that this equation involves the coordinate-space wave function on both the left- and the right-hand sides.

Of course, these manipulations are here carried out for the continuous case, which involves integrals on the right-hand side; you may be wondering why we would bother with Eq. (6.4), since it's replaced one derivative by two integrals. In the present chapter we will learn about a related tool called the *discrete Fourier transform* (DFT) which involves sums. We will then see that it is possible to re-express the DFT so that it is carried out incredibly fast. One of the problems asks you to develop and implement an analogous differentiation formula for the DFT. At that point you will appreciate the power of being able to produce a table of derivative values without having to worry about finite-difference errors. Even more crucially, in the n-dimensional case evaluating the gradient vector at one point via the forward difference requires n new function evaluations, see Eq. (5.103), but Fourier differentiation gives you the derivatives without *any* new function evaluations being necessary.

3. **Hubble's law**
According to observations, the universe displays remarkable large-scale uniformity. More specifically, at large scales the universe is *homogeneous*: this applies not only

to the average density, but also to the types of galaxies, their chemical composition, and other features. Furthermore, the universe appears to be *isotropic* about every point, implying that there is no special direction. Another aspect of the uniformity of the universe is that even its expansion seems to be uniform: on average, galaxies are receding from us with a speed that is proportional to how far away from us they are. Of course, this doesn't apply only to us: any galaxy will see all other galaxies receding at a speed proportional to their distance from it.

The point we just made is significant enough to bear repeating: even though the universe is expanding, the homogeneity and isotropy are maintained, i.e., the universe does not exhibit a velocity anisotropy. This fact was first noticed by Edwin Hubble in 1929, in his study of 24 extra-galactic nebulae for which both distances and velocities were available. Plotting the velocities versus the distances, Hubble noticed a "roughly linear relation" and extracted the coefficient. Nowadays, we call this *Hubble's law*:

$$v = Hd \tag{6.5}$$

where v is the recessional velocity, d is the distance, and H is known as *Hubble's parameter*. The present value of this parameter is known as *Hubble's constant*, and has been measured to be $H_0 = (2.3 \pm 0.1) \times 10^{-18}\ \mathrm{s}^{-1}$. Hubble's parameter has wider significance: if we denote by $R(t)$ the cosmological expansion parameter, then the following relation holds:

$$H = \frac{\dot{R}(t)}{R(t)} \tag{6.6}$$

i.e., at time t Hubble's parameter is the relative rate of expansion of our universe.

In modern terms, Hubble carried out a straight-line fit to the data. We encountered another instance of experimental data following a linear trend earlier on, when we discussed Millikan's 1916 experiment on the photoelectric effect (section 2.1). In the Project at the end of the present chapter we will discuss yet another set of experimental data, this time *not* obeying a linear trend.

It is no coincidence that the first two items on our list both involve taking derivatives. The problem of *approximation*, the theme of this chapter, typically arises when one wishes to carry out operations that are otherwise hard or impossible.

6.1.2 The Problems to Be Solved

Basically, the goal of the present chapter is to learn how to approximate a function $f(x)$. There could be several reasons why this needs to happen: perhaps computing $f(x)$ is a costly procedure, which cannot be repeated at many points, or maybe $f(x)$ can only be experimentally measured. In any case, the point of the present chapter is that we wish to have access to $f(x)$ for general x but can only produce selected $f(x_j)$ values. What is commonly done is that a set of n *basis functions* $\phi_k(x)$ is chosen which, combined with a set of n undetermined parameters c_k ($k = 0, 1, \ldots, n-1$) is used to produce the following *linear form*:

$$p(x) = \sum_{k=0}^{n-1} c_k \phi_k(x) \tag{6.7}$$

This is called a linear form because it constitutes a linear combination of basis functions; this does *not* mean that the basis functions themselves are linear. We assume that the $\phi_k(x)$'s are *linearly independent*. Note that we used $p(x)$ to denote our *approximating function*, since we don't know that it is the same as the underlying $f(x)$. Crucially, both $p(x)$ and the basis $\phi_k(x)$ apply to any x, not just points on a grid. This raises the question of how we would know that our approximating function, $p(x)$, is accurate, leading us to distinguish between the following two large classes of approach:

1. **Interpolation**

   Interpolation arises when we have as input a table of data points, (x_j, y_j) for $j = 0, 1, \ldots, n-1$, which we assume exactly represent the underlying $f(x)$.[1] We'll take the basis functions $\phi_k(x)$ as given and attempt to determine the n parameters c_k. Crucially, we have n unknowns and n data points, so we can determine all the unknown parameters by demanding that our approximating function $p(x)$ go *exactly* through the input data points (except for roundoff error), i.e.:

   $$y_j = \sum_{k=0}^{n-1} c_k \phi_k(x_j) \tag{6.8}$$

   where we used the fact that $p(x_j) = y_j$. This can be written in matrix form as follows:

$$\begin{pmatrix} \phi_0(x_0) & \phi_1(x_0) & \phi_2(x_0) & \ldots & \phi_{n-1}(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \phi_2(x_1) & \ldots & \phi_{n-1}(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \phi_2(x_2) & \ldots & \phi_{n-1}(x_2) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \phi_0(x_{n-1}) & \phi_1(x_{n-1}) & \phi_2(x_{n-1}) & \ldots & \phi_{n-1}(x_{n-1}) \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix} \tag{6.9}$$

   or, more compactly, $\mathbf{\Phi c} = \mathbf{y}$, where $\mathbf{\Phi}$ is an $n \times n$ matrix and we're solving for the $n \times 1$ column vector $\mathbf{c}$. Thus, for a given choice of the $\phi_k(x)$, with the table (x_j, y_j) known, this is "simply" a linear system of equations for the c_k's; it can be solved using, say, Gaussian elimination after $O(n^3)$ operations.

   An example of interpolation along these lines is shown in Fig. 6.1, where the solid curve smoothly connects the data points. Significantly, such an *interpolant* is not unique: even though every interpolant *has* to go through the data points, there are many shapes/forms

[1] Interpolation has been called an "exact approximation", but this is infelicitous (if not a contradiction in terms).

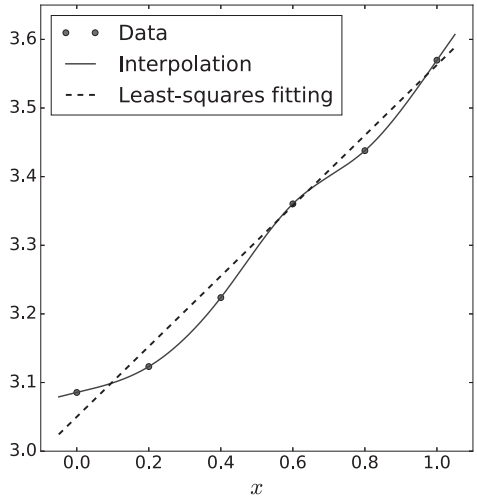


Illustration of the difference between interpolation and least-squares fitting **Fig. 6.1**

it could have taken between them.[2] As a result, one needs to impose further criteria in order to choose an interpolant, such as smoothness, monotonicity, etc. The crudest way to put it is that the interpolant should behave "reasonably"; for example, wild fluctuations that are not supported by the data should generally be avoided.

In the present chapter, we'll examine three different approaches to the interpolants:

(a) **Polynomial interpolation**: this approach, covered in section 6.2, assumes that a single polynomial can efficiently and effectively capture the behavior of the underlying function. Despite popular myths to the contrary, this is a great assumption.

(b) **Piecewise polynomial interpolation**: here, as we'll see in section 6.3, one breaks up the data points into *subintervals* and employs a different low-degree polynomial in each subinterval.

(c) **Trigonometric interpolation**: in section 6.4 we'll find out how to carry out interpolation for the case of periodic data. This will also allow us to introduce one of the most famous algorithms in existence, the *fast Fourier transform*.

Regardless of how one picks the basis functions, having access to an interpolation scheme usually helps one before carrying out further manipulations; for example, one might wish to differentiate or integrate $p(x)$. Assuming the interpolation was successful, such tasks can be carried out much faster and with fewer headaches than before.

2. **Least-squares fitting**

We now turn to our next large class of approximation. There are two major differences from the case of interpolation: (a) we have more data than parameters, and (b) our input

[2] Actually, in Fig. 6.1 we are also showing our interpolant for x values slightly outside the interval where we have data; generally, this is a trickier problem, known as *extrapolation*. In cases where you have a theory about how the data should behave even in regions where you have no measurements, this is not too hard a problem; this is precisely the case of Richardson extrapolation, discussed in chapters 3 and 7.

data have associated errors. Specifically, this time we have as input a table of data points, (x_j, y_j) for $j = 0, 1, \ldots, N-1$, which do *not* perfectly represent the underlying $f(x)$; for example, these could arise in experimental measurement, in which case each y_j is associated with an error, σ_j. Note that in the present case the number of data points, N, is larger than the number of c_k parameters in Eq. (6.7), which was n, i.e., we have $N > n$. In other words, the problem we are faced with is that the input data cannot be fully trusted, i.e., our approximating function should not necessarily go exactly through them, but at the same time we have more data points then we have parameters.

In the language of linear algebra, the problem we are now faced with is an *overdetermined system*, of the form:

$$\begin{pmatrix} \phi_0(x_0) & \phi_1(x_0) & \ldots & \phi_{n-1}(x_0) \\ \phi_0(x_1) & \phi_1(x_1) & \ldots & \phi_{n-1}(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \ldots & \phi_{n-1}(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(x_{N-1}) & \phi_1(x_{N-1}) & \ldots & \phi_{n-1}(x_{N-1}) \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{pmatrix} \approx \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{N-1} \end{pmatrix} \tag{6.10}$$

Here our input y-values are grouped together in an $N \times 1$ column vector $\mathbf{y}$, $\mathbf{\Phi}$ is an $N \times n$ matrix, and we're solving for the $n \times 1$ column vector $\mathbf{c}$. As you can see, we are aiming for approximate equality, since this system cannot be generally solved (and even if it could, we know that our y_j values suffer from errors).

To tackle this overdetermined system, we'll do the next-best thing available; since we can't solve $\mathbf{\Phi c} = \mathbf{y}$, let's try to *minimize the norm of the residual vector*, i.e.,

$$\min \|\mathbf{\Phi c} - \mathbf{y}\| \tag{6.11}$$

where we're implicitly using the Euclidean norm. As we'll see in section 6.5 this approach is known as *least-squares fitting*, since it attempts to determine the unknown vector by minimizing the (sum of the) squares of the difference between the approximating function values and the input data.[3]

An example of least-squares fitting is shown in Fig. 6.1: if we assume that the data points cannot be fully trusted, then it is more appropriate to capture their overall trend. As a result, the goal of our least-squares fitting procedure should not be to go through the points. We'll learn more about how to quantify the effect of the error in the input data later on but, for now, observe that the least-squares fit in Fig. 6.1 follows from the assumption that the approximating function $p(x)$ should be *linear in* x. Obviously, we could have made different assumptions, e.g., that $p(x)$ is quadratic in x, and so on.

[3] Another way of approaching this overdetermined system is via *minimax approximation*, which aims to minimize the maximum error, instead. This is an elegant idea, but covering it would require a separate development.

Apparently, we'll need to introduce a goodness-of-fit measure to help us decide which form to employ.

In section 6.5 we'll first learn about straight-line fits, which involve only two parameters (as in Fig. 6.1); this is the simplest-possible choice of basis functions. We will then turn to the more general problem of tackling the *normal equations* that arise in the case of polynomial least-squares fitting, employing machinery we developed in our study of linear algebra in chapter 4. It's important to realize that even polynomial least-squares fitting is a case of *linear least-squares fitting*: you can see from Eq. (6.7) that the dependence on the c_k parameters would be linear, even if the basis functions are, e.g., exponentials. In the Project of section 6.6 you will get a first exposure to the more general problem of *nonlinear least-squares fitting*; while in general this is a problem of multidimensional minimization, as in section 5.5.2, we will rewrite it in the form of a multidimensional root-finding problem and proceed to employ methods we introduced in section 5.4.[4]

6.2 Polynomial Interpolation

We now turn to our first major theme, namely *polynomial interpolation*; this refers to the case where the interpolating function $p(x)$ of Eq. (6.7) is a polynomial. Keep in mind that the underlying function $f(x)$ that you're trying to approximate does *not* have to be a polynomial, it can be anything whatsoever. That being said, we will study cases that are well-behaved, i.e., we won't delve too deeply into singularities, discontinuities, and so on; in other words, we focus on the behavior that appears most often in practice. As explained in the preceding overview, we start by using a single polynomial to describe an entire interval; in addition to its intrinsic significance, this is a topic which also help us in chapter 7 when we study numerical integration.

At the outset, let us point out that polynomial interpolation is a subject that is misleadingly discussed in a large number of texts, including technical volumes on numerical analysis.[5] The reasons why this came to be are beyond the scope of our text; see N. Trefethen's book on approximation theory [94] for some insights on this and several related subjects. Instead, let us start with the final conclusion: *using a single polynomial to approximate a complicated function is an excellent approach, if you pick the interpolating points x_j appropriately*. This is not just an abstract question: knowing how easy, efficient, and accurate polynomial interpolation can be, will benefit you in very practical ways in the future. Issues with single-polynomial interpolation arise when one employs equidistant points, an example which is all most textbooks touch upon.[6] We'll here follow a pedagogi-

[4] This also helps explain why our discussion of approximating functions was not provided in chapter 2; our exposition will involve the linear algebra and root-finding machinery that we developed in chapters 4 and 5.

[5] *Amicus Plato, sed magis amica veritas*, i.e., Plato is a friend, but truth is a better friend.

[6] The standard treatment, which employs equidistant points and gets into trouble, doesn't mesh well with the *Weierstrass approximation theorem,* which states that a polynomial *can* approximate a continuous function; the problem is that this theorem doesn't tell us how to go about constructing such a successful polynomial.

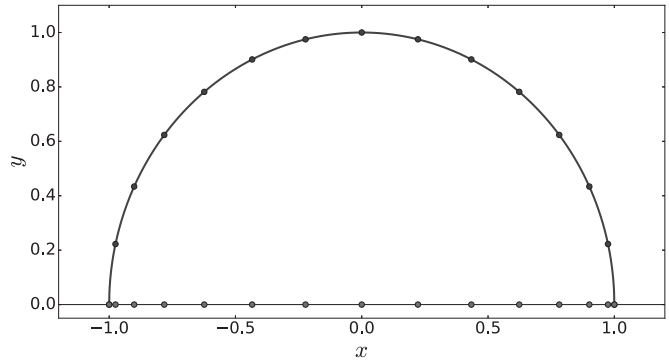


Fig. 6.2 Chebyshev nodes for $n = 15$, also showing the equidistant points on the unit circle

cal approach, i.e., we will start with what works and only mention potential problems near the end of our discussion.

To be specific, let us immediately introduce our go-to solution for the placement of the interpolation nodes. In much of what follows, we employ *Chebyshev points*, sometimes known as *Chebyshev nodes*. As you learned in a problem in chapter 2, see Eq. (2.123), the extrema of *Chebyshev polynomials* take the simple form:

$$x_j = -\cos\left(\frac{j\pi}{n-1}\right), \qquad j = 0, 1, \ldots, n-1 \tag{6.12}$$

where we have included a minus sign in order to count them from left to right.[7] Given the importance of these points, we will mainly focus on the interval $[-1, 1]$; if you are faced with the interval $[a, b]$ you can scale appropriately:

$$t = \frac{b+a}{2} + \frac{b-a}{2}x \tag{6.13}$$

Spend some time to understand how this transformation takes us from x which is in the interval $[-1, 1]$ to t which is in the interval $[a, b]$: $x = -1$ corresponds to $t = a$, $x = 1$ corresponds to $t = b$ and, similarly, the midpoint $x = 0$ corresponds to $t = (b + a)/2$. Note that Eq. (6.13) is more general than Eq. (6.12), i.e., you can use it even if you are not employing Chebyshev points.

In order to help you build some intuition on the Chebyshev points, we are showing them in Fig. 6.2 on the x axis. Also shown are the equidistant points on the (upper half of the) unit circle; their projections onto the x axis are the Chebyshev points. The most conspicuous feature of Chebyshev points is that they cluster near -1 and 1 (i.e., near the ends of the interval). This will turn out to be crucial for the purposes of interpolation, as

[7] Though the convention in the literature is not to include this minus sign.

discussed in section 6.2.3. Despite not being equidistant on the x axis, Chebyshev points have the advantage of obeying *nesting*: when you double the number of points, n, you use, you can re-use the old points since you're simply adding new points between the old ones. This is generally desirable when you don't know ahead of time how many points you are going to need: start with some guess, keep doubling (while re-using the work you carried out for the old guesses), until you see that you can stop doubling.

The attentive reader may recall encountering another set of points with a similar clustering property, namely the roots of *Legendre polynomials*, which we computed all the way back in section 5.3.2. For now, we will consider the placement of the nodes as settled, i.e., following Chebyshev points as per Eq. (6.12). This still leaves open the question of how to solve Eq. (6.9): we need to pick a specific set of basis functions $\phi_k(x)$ and then to compute the c_k parameters.

In order to avoid getting confused, one has to distinguish between the (possible) *ill-conditioning* of a *problem* and the *stability* of a given *algorithm* employed to solve that problem. For example, in section 4.2 we discussed how to quantify the conditioning of a linear system of equations; this had little to do with specific methods one could choose to apply to that problem. If a matrix is terribly ill-conditioned it will be so no matter which algorithm you use on it. Similarly, in section 4.3.4 we encountered a case (Gaussian elimination without pivoting) where the algorithm can be unstable even when the problem is perfectly well-conditioned. Turning to the present case of polynomial interpolation, one has to be careful in distinguishing between the conditioning of the problem itself (interpolating using a table (x_j, y_j)) and the method you employ to do that (which needs to evaluate the c_k parameters). We'll soon see a specific method that can be quite unstable, but this doesn't mean that the problem itself is pathological.

Another distinction one can make at a big-picture level relates to the practical implementation of polynomial interpolation. While we will always follow Eq. (6.7), we can distinguish between two stages in our process: (a) *constructing* the interpolating function is the first stage; after deciding on which basis functions $\phi_k(x)$ to employ, the main task here is to compute the c_k parameters, and (b) *evaluating* the interpolating function $p(x)$ at a given x, with the $\phi_k(x)$'s and c_k's in place. It stands to reason that the first stage, computing the c_k parameters, should only be carried out once, i.e., should be independent of the specific x where we may need to evaluate $p(x)$. As mentioned earlier, interpolation is supposed to make further manipulations easy, so the evaluation stage should be as efficient as possible; this would not be the case if the parameter values would need to be separately re-computed for each new x point.

6.2.1 Monomial Basis

The preceding discussion may be too abstract for your taste, so let us write down some equations. The most natural choice for the basis functions $\phi_k(x)$ is to use monomials:[8]

$$\phi_k(x) = x^k \tag{6.14}$$

[8] You may recall monomials as what you started with before orthogonalizing to get to Legendre polynomials, in a problem in chapter 4.

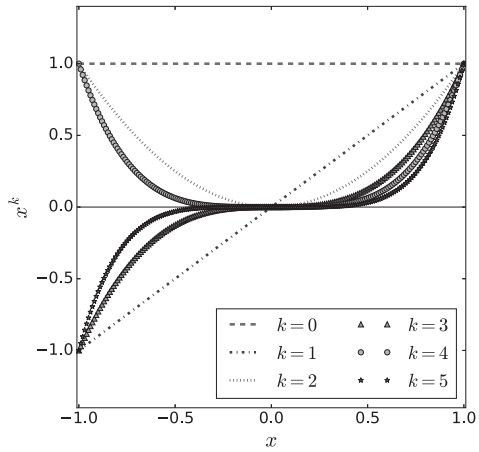


Fig. 6.3 Monomial basis functions for the first several degrees

It's not too early to highlight the fact that we use j for the spatial index and k for the index that keeps track of the basis functions. Plugging Eq. (6.14) into Eq. (6.7) allows you to write the interpolating polynomial $p(x)$ in the simplest way possible:

$$p(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \cdots + c_{n-1} x^{n-1} \tag{6.15}$$

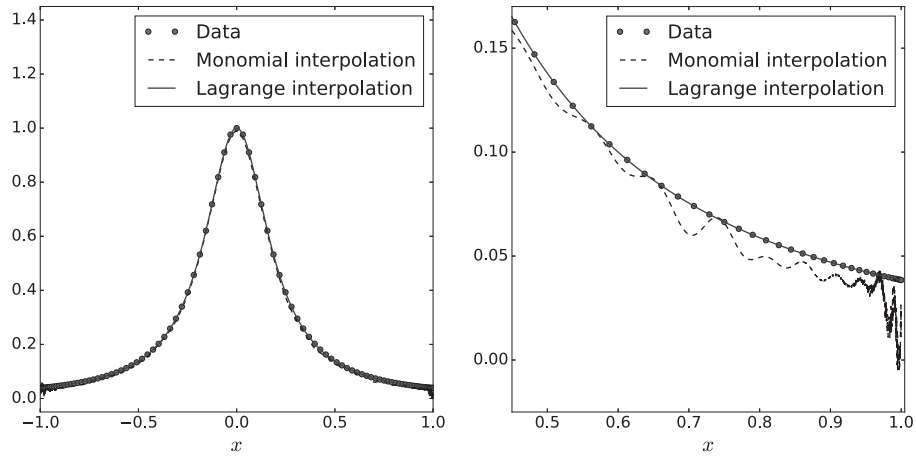
This employs notation similar to that we've used before, say in Eq. (5.62): this is a polynomial of degree $n-1$, in keeping with our 0-indexing convention. Remember, our input is a table of data points, (x_j, y_j) for $j = 0, 1, \ldots, n-1$; we have n unknown parameters and n input points.

The most naive way to carry out polynomial interpolation, which is the one you probably learned when you first saw how to find the line that goes through two specified points, is to evaluate Eq. (6.15) at our grid points:

$$p(x_j) = y_j = c_0 + c_1 x_j + c_2 x_j^2 + c_3 x_j^3 + \cdots + c_{n-1} x_j^{n-1} \tag{6.16}$$

and solve for the unknown parameters. Explicitly writing this out for all the values of j, i.e., $j = 0, 1, \ldots, n-1$, allows us to re-express our equations in matrix form:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix} \tag{6.17}$$



Monomial interpolation for the case of $n = 101$ (left), also zoomed in (right) **Fig. 6.4**

This (nonsymmetric) $n \times n$ coefficient matrix is known as a *Vandermonde matrix*.[9] As you will show in one of the problems, the determinant of a Vandermonde matrix is non-zero for distinct nodes x_j; thus, a Vandermonde matrix is *non-singular*. As a result, the columns of the Vandermonde matrix are linearly independent. This means that if you employ $O(n^3)$ operations (at most, since specialized algorithms also exist) you are guaranteed to be able to solve this linear system of equations for the c_k parameters; this is a *unique* solution. This is a general result: for real data points with distinct nodes, there exists a unique polynomial that goes through the points.

While the fact of the non-singularity of the Vandermonde matrix is good news, it's not great news. In Fig. 6.3 we show the monomial basis functions for the first several degrees. As the degree increases, different basis functions become closer in value to each other for positive-x; for negative-x odd degrees start to look alike, and similarly for even degree monomials. Even though in principle the columns of a Vandermonde matrix are linearly independent, in practice they are almost linearly dependent. Put another way, the Vandermonde matrix becomes increasingly ill-conditioned as n gets larger. In the past, textbooks would brush this off as a non-issue, since one didn't employ interpolating polynomials of a high degree anyway. However, as we're about to see, polynomials with $n = 100$, or even $n = 1000$, are routinely used in practice and therefore the Vandermonde ill-conditioning is a real problem.

Figure 6.4 shows data points drawn from a significant example, *Runge's function*:

$$f(x) = \frac{1}{1 + 25x^2} \tag{6.18}$$

In this figure, our data (x_j, y_j) are picked from the Chebyshev formula in Eq. (6.12) for this specific $f(x)$; we used $n = 101$ to stress the fact that n does not need to be small in practical

[9] This is not the first Vandermonde matrix we've encountered: in one of the problems in chapter 5, see Eq. (5.128), we solved a multidimensional system of nonlinear equations, with the promise that it will re-emerge in our study of Gaussian quadrature in chapter 7.

applications. Note that our interpolation using the monomial basis/Vandermonde matrix does a reasonable job capturing the large-scale features of Runge's function (see left panel). This is starting to support our earlier claim that polynomial interpolation with Chebyshev nodes is an excellent tool. Unfortunately, once we zoom in (see right panel) we realize that the monomial interpolation misses detailed features and shows unpleasant wiggles near the ends of the interval, as a result of the Vandermonde matrix's ill-conditioning. Crucially, this is not an issue with the Runge function; it is also not a problem with the Chebyshev nodes; it *is* a problem with the method we chose to use in order to interpolate between our Chebyshev nodes for the Runge function (though other functions would show similar behavior). In other words, the issue lies with the "naive" choice to use monomials as our basis functions and then solve an ill-conditioned linear system. To emphasize the fact that the polynomial-interpolation problem itself was *not* ill-conditioned (i.e., our algorithm is unstable but we could have picked a better algorithm, instead), we also show in Fig. 6.4 results from a mysterious "Lagrange interpolation" method. Having piqued your curiosity, this is the topic we turn to next.

6.2.2 Lagrange Interpolation

Lagrange interpolation[10] is introduced in most textbooks on numerical methods. After some introductory comments, though, the approach is usually abandoned for other methods, e.g., Newton interpolation, typically after a claim that the Lagrange form is nice for proving theorems but not of practical value. As we'll see, Lagrange interpolation is helpful both formally and in practice. To emphasize this point, we will end the section with an implementation of Lagrange interpolation which leads to the (excellent) results in Fig. 6.4.[11]

Cardinal Polynomials

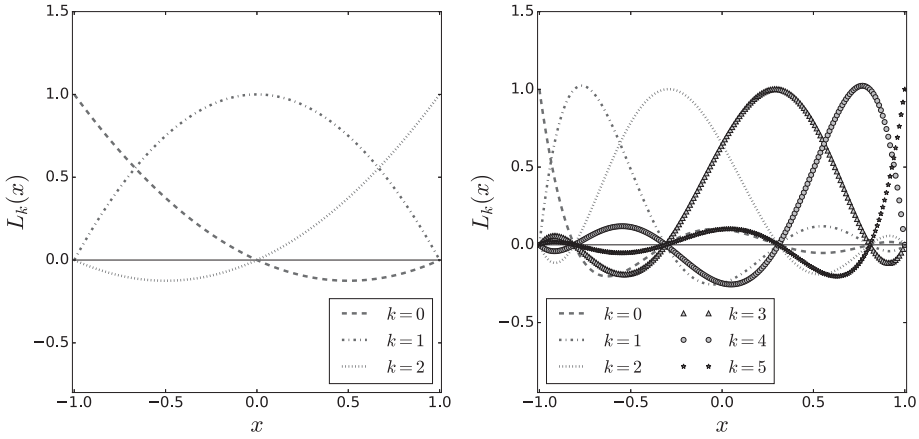
As before, we have as input a table of data points, (x_j, y_j) for $j = 0, 1, \ldots, n-1$. Before discussing how to interpolate through those points, let us introduce what are known as *cardinal* or *Lagrange* or *fundamental* polynomials:

$$L_k(x) = \frac{\prod_{j=0, j\neq k}^{n-1}(x - x_j)}{\prod_{j=0, j\neq k}^{n-1}(x_k - x_j)}, \qquad k = 0, 1, \ldots, n-1 \tag{6.19}$$

The denominator depends only on the x_j's, i.e., on the interpolation points, so it is clearly a constant (it doesn't depend on x). The numerator is a polynomial in x of degree $n-1$, which for a given k goes to 0 at x_j when $j \neq k$. The Lagrange polynomial $L_k(x)$ goes to 1 at x_k, since the numerator and the denominator are equal to each other in that case.

[10] A typical example of the *Matthew effect*: Lagrange interpolation was actually discovered by Waring.

[11] Incidentally, in section 5.3.1 we cautioned against finding the roots of a polynomial starting from its coefficients. Here, however, we're not finding roots but interpolatory behavior; also, we're starting from interpolation nodes, not from polynomial coefficients.



Lagrange basis functions for Chebyshev 3-point (left) and 6-point (right) interpolation **Fig. 6.5**

If you've never encountered Lagrange polynomials before, you might benefit from seeing them explicitly written out for a simple case. Regardless of where the x_j's are placed, here's what the three Lagrange polynomials for the case of $n = 3$ look like:

$$L_0(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)}, \quad L_1(x) = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)}, \quad L_2(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} \tag{6.20}$$

We could expand the parentheses here, but we won't, since that's less informative than the form the fractions are currently in. Clearly, for $n = 3$ the Lagrange polynomials are quadratic in x. Equally clearly, $L_0(x_1) = 0$, $L_0(x_2) = 0$, and $L_0(x_0) = 1$, and analogous relations hold for $L_1(x)$ and $L_2(x)$. These facts are illustrated in the left panel of Fig. 6.5: in the interval $[-1, 1]$ for $n = 3$ Chebyshev points and equidistant points are the same.

As mentioned before, these are general results: $L_k(x_j)$ is 0 if $j \neq k$, and 1 if $j = k$. This result is important enough that it deserves to be highlighted:

$$L_k(x_j) = \delta_{kj} \tag{6.21}$$

where δ_{kj} is the *Kronecker delta*. The right panel of Fig. 6.5 shows the example of $n = 6$, in which case the $L_k(x)$ are fifth degree polynomials (and there are six distinct Lagrange polynomials). No matter what they may do between one node and the next, these always go to 1 and to 0 in the appropriate places.

It is now time to use our cardinal polynomials as *basis functions*; this means that we take $\phi_k(x) = L_k(x)$ in Eq. (6.7). As will soon become clear, this also means that we don't need to use c_k in our definition of the interpolating polynomial, since we can simply write y_k (the y-values of our input data) in their place:

$$p(x) = \sum_{k=0}^{n-1} y_k L_k(x) \tag{6.22}$$

To see why this is allowed, let's examine the value of $p(x)$ at our nodes:

$$p(x_j) = \sum_{k=0}^{n-1} y_k L_k(x_j) = \sum_{k=0}^{n-1} y_k \delta_{kj} = y_j \tag{6.23}$$

where we used Eq. (6.21) in the second step. But $p(x_j) = y_j$ was the definiton of what it means to be an interpolating polynomial.

We thereby conclude that Eq. (6.22) has been successful: by using cardinal polynomials (which go to 0 and to 1 in the right places) instead of monomials, we have managed to avoid encountering a (possibly ill-conditioned) Vandermonde matrix. As a matter of fact, the Lagrange form employs an ideally conditioned basis: in the language of Eq. (6.9), since $\phi_k(x_j) = L_k(x_j) = \delta_{kj}$, we see that $\mathbf{\Phi}$ is the identity matrix. Thus, we don't even have to solve $\mathbf{\Phi c} = \mathbf{y}$, since we'll always have $c_j = y_j$; this explains why we used y_k in Eq. (6.22). To summarize, in the Lagrange form we write the interpolating polynomial as a linear combination (not of monomials, but) of polynomials of degree $n - 1$, see Eq. (6.22).

Barycentric Formula

At this point, you can understand why many books shun the Lagrange form: while it is true that it leads to a linear system in Eq. (6.9) which is perfectly conditioned, the approach still seems to be inefficient. In other words, while it is true that we don't have to solve a linear system, which in general costs $O(n^3)$ operations, our main formula in Eq. (6.22), combined with Eq. (6.19), appears to be highly inefficient: at a single x, we need $O(n)$ operations to compute $L_k(x)$ and must then carry out n such evaluations, leading to a total of $O(n^2)$ operations. When you change the value of x where you need to find the value of the interpolating polynomial, you'll need another $O(n^2)$ operations. To compare with the monomial basis: there it was costly to determine the c_k's as per Eq. (6.17) but, once you had done so, it took only $O(n)$ operations to evaluate $p(x)$ at a new x as per Eq. (6.15).

One can do much better, simply by re-arranging the relevant formulas. Observe that, as you change k, the numerator in Eq. (6.19) is always roughly the same, except for the missing $x - x_k$ factor each time. This motivates us to introduce the following *node polynomial*:

$$L(x) = \prod_{j=0}^{n-1} (x - x_j) \tag{6.24}$$

where there are no factors missing; this $L(x)$ is a (monic) polynomial of degree n. Clearly, $L(x_k) = 0$ for any k, i.e., the node polynomial vanishes at the nodes.

We now make a quick digression, which will not be used below but will help us in the following chapter, when we introduce Gauss–Legendre integration. The numerator in the

equation for $L_k(x)$, Eq. (6.19), can be re-expressed as follows:

$$\prod_{j=0, j\neq k}^{n-1} (x - x_j) = \frac{\prod_{j=0}^{n-1}(x - x_j)}{x - x_k} = \frac{L(x)}{x - x_k} \tag{6.25}$$

where in the second equality we identified the node polynomial from Eq. (6.24). We now turn to the denominator in Eq. (6.19). This can be arrived at using Eq. (6.25), where we write the numerator as $L(x) - L(x_k)$ (since $L(x_k) = 0$) and then take the limit $x \to x_k$. This leads to:

$$\prod_{j=0, j\neq k}^{n-1} (x_k - x_j) = L'(x_k) \tag{6.26}$$

where the definition of the derivative appeared organically. Putting the last two equations together with Eq. (6.19) allows us to express a given cardinal polynomial $L_k(x)$ in terms of the node polynomial and its derivative:

$$L_k(x) = \frac{L(x)}{(x - x_k)L'(x_k)} \tag{6.27}$$

We will produce another (analogous) relation below.

Continuing with the main thread of our argument, we now introduce another quantity that captures the denominator in Eq. (6.19); we define the *barycentric weights* as follows:

$$w_k = \frac{1}{\prod_{j=0, j\neq k}^{n-1}(x_k - x_j)}, \qquad k = 0, 1, \ldots, n-1 \tag{6.28}$$

We are now in a position to re-express (once again) a given cardinal polynomial $L_k(x)$, see Eq. (6.19), in terms of the node polynomial and the weights:

$$L_k(x) = L(x)\frac{w_k}{x - x_k} \tag{6.29}$$

where we simply divided out the $x - x_k$ factor that is present in $L(x)$ but missing in $L_k(x)$.

A way to make further progress is to remember that, regardless of how you compute $L_k(x)$, the way to produce an interpolating polynomial $p(x)$ is via Eq. (6.22). Let us write out that relation, incorporating Eq. (6.29):

$$p(x) = L(x)\sum_{k=0}^{n-1} y_k \frac{w_k}{x - x_k} \tag{6.30}$$

where we pulled out $L(x)$ since it didn't depend on k. We now apply Eq. (6.22) again, this time for the special case where all the y_j's are 1: given the uniqueness of polynomial interpolants, the result must be the constant polynomial 1. In equation form, we have:

$$1 = \sum_{k=0}^{n-1} L_k(x) = L(x)\sum_{k=0}^{n-1} \frac{w_k}{x - x_k} \tag{6.31}$$

In the first step we show that adding up all the Lagrange polynomials gives us 1; in the

second step we plugged in Eq. (6.29). If we now divide the last two equations, we can cancel the node polynomial $L(x)$, leading to:

$$p(x) = \sum_{k=0}^{n-1} \frac{w_k y_k}{x - x_k} \Bigg/ \sum_{k=0}^{n-1} \frac{w_k}{x - x_k} \tag{6.32}$$

where we get the w_k from Eq. (6.28). If you run into the special case $x = x_k$ for some k, you can simply take $p(x) = y_k$. Equation (6.32) is known as the *barycentric interpolation formula* and is both a beautiful and a practical result. Observe that the two sums are nearly identical: the first one involves the input data values y_k, whereas the second one doesn't. As x approaches one of the x_k's, a single term in the numerator and in the denominator become dominant; their ratio is y_k, so our interpolating polynomial has the right behavior. Of course, if x is actually equal to x_k this line of reasoning doesn't work, which is why we account for this scenario separately in our definition.

You may be wondering why we went through all the trouble. The form in Eq. (6.22) was clearly a polynomial, whereas Eq. (6.32), despite still being a polynomial, looks like it might be a rational function. The reason we bothered has to do with our earlier complaint that the Lagrange form of Eq. (6.22) necessitated $O(n^2)$ operations at each x value. In contradistinction to this, the barycentric formula splits the *construction stage* from the *evaluation stage*: we first compute the weights as per Eq. (6.28), using $O(n^2)$ operations; crucially, the weights don't rely on the y_j values but only on the placement of the nodes, x_j. When the weights have been computed (once and for all), we use Eq. (6.32) at each new x, requiring only $O(n)$ operations. As you may recall, this was the scaling exhibited by the monomial-basis evaluation stage, only this time we have no ill-conditioning issues.

Even though Eq. (6.32) looks like it might have issues with roundoff error, it is numerically stable. It actually has several more advantages, e.g., in how it handles the introduction of a new data pair (x_n, y_n). Another pleasant feature is that for Chebyshev nodes the weights w_k can be analytically derived, i.e., the $O(n^2)$ operations are not needed. Instead of getting into such details, however, we will now see the barycentric formula in action.

Implementation

We will study Runge's function from Eq. (6.18), namely the same example we plotted for the case of the monomial basis. In reality, interpolation deals with a table of data (x_j, y_j) as input, not with continuous functions. Even so, we want to test how well our interpolation is working, so it's good to start from a function and *produce* our data from it. Thus, Code 6.1 starts by defining Runge's function and then introduces another function to generate the data pairs. We have coded up two distinct possibilities: Chebyshev points as per Eq. (6.12), or equidistant nodes. If you're still not very comfortable with `numpy.linspace()`, try to use `numpy.arange()` to accomplish the same task; for example, for Chebyshev nodes you would write down something like `np.arange(n)*np.pi/(n-1)`. Note that `dataxs` is a `numpy` array, so when we define `datays` we end up calling `f()` with an array as an

barycentric.py Code 6.1

```
import numpy as np

def f(x):
     return 1/(1 + 25*x**2)

def generatedata(n,f,nodes="cheb"):
     if nodes=="cheb":
          dataxs = -np.cos(np.linspace(0,np.pi,n))
     else:
          dataxs = np.linspace(-1,1,n)
     datays = f(dataxs)
     return dataxs, datays

def weights(dataxs):
     n = dataxs.size
     ws = np.ones(n)
     for k in range(n):
          for j in range(n):
               if j == k:
                    continue
               ws[k] *= (dataxs[k]-dataxs[j])
     return 1/ws

def bary(dataxs,datays,ws,x):
     k = np.where(x == dataxs)[0]
     if k.size == 0:
          nume = np.sum(ws*datays/(x-dataxs))
          denom = np.sum(ws/(x-dataxs))
          val = nume/denom
     else:
          val = datays[k[0]]
     return val

if __name__ == '__main__':
     dataxs, datays = generatedata(15, f)
     ws = weights(dataxs)
     x = 0.3; pofx = bary(dataxs, datays, ws, x)
     print(x, pofx, f(x))
```

argument; this works seamlessly for the operations involved in defining Runge's function (power, multiplication, addition, and division).

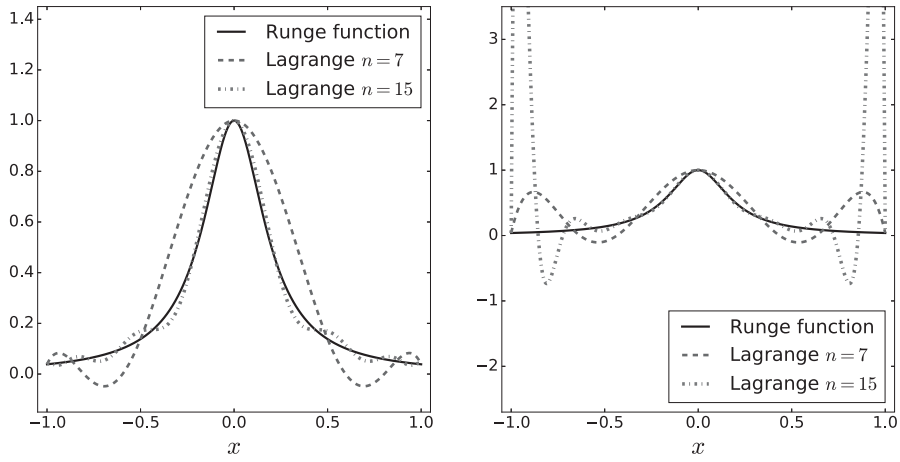
Having produced the table of data (x_j, y_j), we need to carry out the actual interpolation procedure. As hinted at above, we do this in two separate steps. We first define a function that computes the weights w_k; this knows nothing about the y_j values: if you need to carry out interpolation for a different problem later on, you can re-use the same weights. Similarly, our computation of the weights is not related to the specific x where we wish to interpolate: it only cares about where the nodes are located, as per Eq. (6.28). You can clearly see that this, construction, stage involves $O(n^2)$ operations, since it contains two nested loops. We eliminate the case $j = k$ from the product by employing `continue`, our first use of this keyword after our introduction to Python at the start of the book. Employing `numpy` functionality, we produce the denominator of w_k for each case, and then invert them all at once at the end.

With the table (x_j, y_j) and the weights w_k in place, we are in a position to turn to the second stage of the interpolation process, namely to use Eq. (6.32) in order to produce $p(x)$ at a given x. We start by considering the special case $x = x_k$ for some k, for which we should take $p(x) = y_k$ instead of using the full barycentric formula. We accomplish this by using `numpy.where()`; as you may recall from our introduction to `numpy`, `numpy.where()` allows us to find specific indices where a condition is met. Since it returns a *tuple* of arrays, we take its 0th element. If the element we're looking for is not there, the 0th element of the tuple returned by `numpy.where()` is an empty array; this means that its `size` is 0, so we should proceed with Eq. (6.32). That case is a straightforward application of `numpy` functionality: we carry out the two needed sums without employing any explicit loops. You might be wondering about the cost of traversing the array elements twice (once for the numerator and once for the denominator); the thing to note is that the evaluation of our interpolant is now a problem requiring $O(n)$ operations, so a factor of 2 doesn't make much of a difference. If `numpy.where()` did *not* return an empty array, we are dealing with the special case, so we index once again in order to return $p(x) = y_k$.

The main program produces the data, constructs the weights, and then evaluates the interpolating polynomial $p(x)$. In order to effectively distinguish the x_j (part of our input data) from the x (the point at which we wish to interpolate), we are calling the former `dataxs` and the latter `x`. The specific value of x we employ here is not important; the output of running this code is:

```
0.3 0.259275058184 0.3076923076923077
```

Since we have access to the underlying $f(x)$, we also compare its value to $p(x)$. The match is not great, so we then decide to use this code at thousands of points to produce a quasi-continuous curve. It's important to realize that we are not referring to the use of thousands of nodes (though we could have done that, as well): we pick, say, $n = 15$ Chebyshev nodes and then plot $p(x)$ for thousands of different x's. The result is shown in the left panel of Fig. 6.6 for both $n = 7$ and $n = 15$. In order to make this plot easier to understand, we are not showing the data points (x_j, y_j). We see that $n = 7$ captures only the peak height, but misses the overall width of our curve, in addition to exhibiting strong oscillations. For



Lagrange interpolation for Chebyshev (left) and equidistant nodes (right) Fig. 6.6

$n = 15$ the width is better reproduced and the magnitude of the oscillations is considerably reduced. As you may recall from our comparison with the monomial basis, see Fig. 6.4, you can keep increasing n and you will keep doing a better job at approximating the underlying $f(x)$; those two panels showed that for $n = 101$ the match is nearly perfect. This cannot be over-emphasized: the canonical problem of misbehavior in polynomial interpolation (Runge's function) is easy to handle using Chebyshev points and the barycentric formula. As noted above, without the barycentric formula we would have had to keep re-evaluating the $L_k(x)$ from scratch at each of our thousands of x values, so this would have been prohibitively slow. To summarize, the accuracy resulted from the Chebyshev points and the efficiency from the barycentric formula.

We left the worst for last. If you use our code for equidistant nodes, you will find behavior like that shown in the right panel of Fig. 6.6. As n is increased, the width of our curve is captured better, but the magnitude of the oscillations at the edges gets *larger*! Even though the interpolating polynomial $p(x)$ goes through the input data points, the behavior near the ends of the interval is wholly unsatisfactory. It is this poor behavior that has often led to the generalization that polynomial interpolation should be avoided when n is large; you now know that this only applies to equidistant nodes. Of course, you may be in the unfortunate situation where the input data were externally provided at equidistant points; in that scenario, one typically resorts to a cubic-spline interpolation instead, as we will see in section 6.3. Before we get there, though, let's build some intuition into why different placements of the nodes lead to such dramatically different results.

6.2.3 Error Formula

As we just saw, polynomial interpolation behaves well for Chebyshev nodes and not so well for equidistant nodes. Instead of examining things on a case-by-case basis, it would be nice to see how well our interpolating polynomial $p(x)$ approximates the underlying $f(x)$ in general. The answer is trivial in the case where $x = x_j$: the interpolant goes through

the nodes, i.e., $p(x_j) = y_j = f(x_j)$. In other words, the *interpolation error* is zero at the nodes. Motivated by this fact, we introduce a new function:

$$F(x) = f(x) - p(x) - L(x)K = f(x) - p(x) - K\prod_{j=0}^{n-1}(x - x_j) \tag{6.33}$$

where K is a constant and in the second equality we plugged in the node polynomial from Eq. (6.24). It's easy to see that if x is on one of the nodes the right-hand-side vanishes, since there the interpolant matches the underlying function. We introduced K because it also helps us see how close $p(x)$ gets to matching $f(x)$ for the more interesting case where x is not equal to the nodes. Let us demand $F(x^*) = 0$ at a given point x^*; solving for K gives us:

$$K = \frac{f(x^*) - p(x^*)}{L(x^*)} \tag{6.34}$$

where we're allowed to divide with $L(x^*)$ since we know that it vanishes only at the nodes. In what follows we won't need this form, so we simply write K.

We have learned that $F(x)$ has n zeros at the x_j's and another zero at x^*. Thus, $F(x)$ has at least $n + 1$ zeros in the interval $[a, b]$. We now make use of *Rolle's theorem*, which you may recall from basic calculus as a special case of the *mean-value theorem*; Rolle's theorem tells us that between every two consecutive zeros of $F(x)$ there is a zero of $F'(x)$. Since $F(x)$ has at least $n+1$ zeros in our interval, we see that $F'(x)$ will have at least n zeros in the same interval. Similarly, $F''(x)$ will have at least $n - 1$ zeros, and so on. Repeatedly applying Rolle's theorem eventually leads to the result that $F^{(n)}(x)$ (the n-th derivative of $F(x)$) has at least one zero in the interval $[a, b]$; if we denote this zero by ξ, our finding is that $F^{(n)}(\xi) = 0$.

Let us examine the n-th derivative a bit more closely. Using Eq. (6.33), we have:

$$F^{(n)}(x) = f^{(n)}(x) - p^{(n)}(x) - L^{(n)}(x)K \tag{6.35}$$

Now, recall from our discussion of Eq. (6.22) that $p(x)$ is a linear combination of polynomials of degree $n - 1$, so it is itself a polynomial of degree at most $n - 1$. This implies that $p^{(n)}(x) = 0$. Similarly, we can see from the definition of the node polynomial in Eq. (6.24) that $L(x)$ is a polynomial of degree n. Actually, it happens to be a *monic polynomial*: as explained near Eq. (5.69), this is what we call a polynomial whose leading power has a coefficient of 1. The fact that $L(x)$ is a monic polynomial makes it very easy to evaluate its n-th derivative: $L^{(n)}(x) = n!$ is the straightforward answer. Putting everything together:

$$F^{(n)}(\xi) = f^{(n)}(\xi) - Kn! = 0 \tag{6.36}$$

where $F^{(n)}(\xi) = 0$, since we're examining its zero. This equation can now be solved for K:

$$K = \frac{f^{(n)}(\xi)}{n!} \tag{6.37}$$

This is a more useful result than Eq. (6.34).

We are now in a position to take the expression for the constant from Eq. (6.37) and combine it with our earlier result $f(x^*) = p(x^*) + L(x^*)K$, to give:

$$f(x^*) = p(x^*) + \frac{f^{(n)}(\xi)}{n!}\prod_{j=0}^{n-1}(x^* - x_j) \tag{6.38}$$

where we've assumed all along that $f(x)$ has an n-th derivative. This is our desired general *error formula* for polynomial interpolation. It shows us that the error in approximating the underlying function is made up of two parts: the n-th derivative of the underlying function (at the unknown point ξ, divided by $n!$) and the node polynomial.

Of the two terms making up the error, the first one, $f^{(n)}(\xi)$, is complicated to handle, since ξ depends on the x_j's implicitly. It's easier to focus on the second term, namely the node polynomial. In a problem, you will see that Chebyshev points minimize the relevant product and also discover how differently equidistant nodes behave; this provides retroactive justification for our choice to employ Chebyshev nodes. Of course, in some cases the $f^{(n)}(\xi)$ will also play a role: in general, the smoother the function $f(x)$ is, the faster the interpolation will converge; the extreme case is Lagrange interpolation at Chebyshev points for *analytic* functions, in which case the interpolant converges geometrically, as we'll see in section 6.3.3. That being said, you don't necessarily have to limit yourself to Chebyshev nodes: another problem asks you to use the roots of *Legendre polynomials*, which as you'll find out also do a good job. What makes both sets of nodes good is the fact that they cluster near the ends of the interval; as a result, each node has roughly the same average distance from the others, something which clearly distinguishes them from equidistant points.

6.2.4 Hermite Interpolation

In the previous subsections we have been tackling the problem of interpolating through a set of data points, i.e., $p(x_j) = y_j$ for $j = 0, 1, \ldots, n-1$. A slightly different problem involves interpolating through both the points *and* the first derivatives at those points:

$$p(x_j) = y_j, \qquad p'(x_j) = y'_j, \qquad j = 0, 1, \ldots, n-1 \tag{6.39}$$

The scenario involving these $2n$ conditions gives rise to what is known as *Hermite interpolation*. As you can guess, using a polynomial of degree $n-1$ won't work: this would have n undetermined parameters, but we need to satisfy $2n$ conditions. In other words, our Ansatz for the interpolating polynomial should start from a form containing $2n$ undetermined parameters, i.e., a polynomial of degree $2n-1$.

We start from the following guess:

$$p(x) = \sum_{k=0}^{n-1} y_k \alpha_k(x) + \sum_{k=0}^{n-1} y'_k \beta_k(x) \tag{6.40}$$

where the $\alpha_k(x)$ are meant to capture the function values and the $\beta_k(x)$ the derivative values.

In other words:

$$\begin{aligned} \alpha_k(x_j) &= \delta_{kj} & \beta_k(x_j) &= 0 \\ \alpha_k'(x_j) &= 0 & \beta_k'(x_j) &= \delta_{kj} \end{aligned} \tag{6.41}$$

for $k, j = 0, 1, \ldots, n-1$. This is merely re-stating our $2n$ conditions from Eq. (6.39).

We decide to write down $\alpha_k(x)$ and $\beta_k(x)$ in terms of $L_k(x)$, namely the cardinal polynomials of Eq. (6.19). As you may recall, $L_k(x)$ is a polynomial of degree $n-1$ which satisfies $L_k(x_j) = \delta_{kj}$, as per Eq. (6.21). We know that we need $\alpha_k(x)$ and $\beta_k(x)$ to be of degree $2n-1$. With that in mind, we realize that if we square $L_k(x)$ we get a polynomial of degree $2(n-1) = 2n-2$. Thus, multiplying $L_k^2(x)$ with a linear polynomial brings us up to degree $2n-1$, as desired:

$$\alpha_k(x) = u_k(x)L_k^2(x), \quad \beta_k(x) = v_k(x)L_k^2(x), \quad k = 0, 1, \ldots, n-1 \tag{6.42}$$

where both $u_k(x)$ and $v_k(x)$ are linear. To see how these are determined, we first write out two properties of the squared cardinal polynomials:

$$\begin{aligned} L_k^2(x_j) &= (\delta_{kj})^2 = \delta_{kj} \\ (L_k^2(x_j))' &= 2L_k(x_j)L_k'(x_j) = 2\delta_{kj}L_k'(x_j) \end{aligned} \tag{6.43}$$

We can now examine the first $\alpha_k(x)$-related condition in Eq. (6.41):

$$\delta_{kj} = \alpha_k(x_j) = u_k(x_j)L_k^2(x_j) = u_k(x_j)\delta_{kj} \tag{6.44}$$

where we used Eq. (6.43) in the last step. This implies that:

$$u_k(x_k) = 1 \tag{6.45}$$

Similarly, the next $\alpha_k(x)$-related condition in Eq. (6.41) gives us:

$$0 = \alpha_k'(x_j) = u_k'(x_j)L_k^2(x_j) + u_k(x_j)(L_k^2(x_j))' = u_k'(x_j)\delta_{kj} + 2u_k(x_j)\delta_{kj}L_k'(x_j) \tag{6.46}$$

where we, again, used Eq. (6.43) in the last step. This implies that:

$$u_k'(x_k) + 2u_k(x_k)L_k'(x_k) = 0 \tag{6.47}$$

Since $u_k(x)$ is linear, we know that $u_k(x) = \gamma x + \delta$. Thus, we can use Eq. (6.45) and Eq. (6.47) to determine γ and δ, in which case we have fully determined $\alpha_k(x)$ to be:

$$\alpha_k(x) = \Big[1 + 2L_k'(x_k)(x_k - x)\Big] L_k^2(x) \tag{6.48}$$

A completely analogous derivation allows you to determine $\beta_k(x)$. Putting everything together, our interpolating polynomial from Eq. (6.40) is then:

$$p(x) = \sum_{k=0}^{n-1} y_k \Big[1 + 2L_k'(x_k)(x_k - x)\Big] L_k^2(x) + \sum_{k=0}^{n-1} y_k' \, (x - x_k)L_k^2(x) \tag{6.49}$$

Even if you didn't follow the derivation leading up to this equation, you should make sure to convince yourself that Eq. (6.49) satisfies our $2n$ conditions from Eq. (6.39).

Note that, using techniques similar to those in section 6.2.3, in a problem you will derive the following general *error formula* for Hermite interpolation:

$$f(x^*) = p(x^*) + \frac{f^{(2n)}(\xi)}{2n!} \prod_{j=0}^{n-1} (x^* - x_j)^2 \tag{6.50}$$

This looks very similar to Eq. (6.38) but, crucially, it contains a square on the right-hand side, as well as a $2n$-th derivative.

6.3 Cubic-Spline Interpolation

In the previous section, we covered interpolation using a single polynomial for an entire interval, staying away from the topic of how to deal with discontinuities. These are typically handled using *piecewise polynomial interpolation*, which is also useful when the placement of the nodes is irregular or, as mentioned earlier, when the nodes have been externally determined to be equidistant.

To be specific, our problem is the same as for Lagrange interpolation, namely we are faced with a table of input data points, (x_j, y_j) for $j = 0, 1, \ldots, n-1$. We wish to interpolate between these points, i.e., produce an easy-to-evaluate and well-behaved interpolant that can be computed at any x value. The approach known as *spline interpolation*, named after the thin strips used in building construction, cuts up the full interval into distinct panels, just like we did in section 3.3.6. A low-degree polynomial is used for each subinterval, i.e., $[x_0, x_1]$, $[x_1, x_2]$, and so on. To make things transparent, we will employ the notation:

$$p(x) = s_{k-1,k}(x), \qquad x_{k-1} \le x \le x_k, \qquad k = 1, 2 \ldots, n-1 \tag{6.51}$$

where $s_{k-1,k}(x)$ is the low-degree polynomial that is used (only) for the subinterval $[x_{k-1}, x_k]$; note that k starts at 1 and ends at $n-1$. These pieces are then stitched together to produce a continuous global interpolant; nodes are sometimes known as *knots* or *break points*.

One of the problems sets up piecewise-linear interpolation, but in what follows we focus on the most popular case, namely piecewise-cubic interpolation, known as *cubic-spline interpolation*. This leads to an interpolant that is smooth in the first derivative and continuous in the second derivative. Before deriving the general case of n nodes, we go over an explicit example, that of $n = 3$, to help you build some intuition about how this all works.

6.3.1 Three Nodes

For $n = 3$ we have only three input-data pairs, (x_0, y_0), (x_1, y_1), and (x_2, y_2), so we are faced with only two panels, $[x_0, x_1]$ and $[x_1, x_2]$. In each of these subintervals, we will

use a different cubic polynomial; to make things easy to grasp, we write these out in the monomial basis; that's not the best approach, but it's good enough to start with.

The first panel will be described by the cubic polynomial:

$$s_{0,1}(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3 \tag{6.52}$$

which is exactly of the form of Eq. (6.15). For the second panel we use a separate polynomial; this means that it will have distinct coefficients:

$$s_{1,2}(x) = d_0 + d_1 x + d_2 x^2 + d_3 x^3 \tag{6.53}$$

As was to be expected, each cubic polynomial comes with four unknown parameters, so we are here faced with eight parameters for our two cubic polynomials put together. We'll need eight equations to determine the unknown parameters.

The first polynomial should interpolate the data at the ends of its (sub)interval:

$$c_0 + c_1 x_0 + c_2 x_0^2 + c_3 x_0^3 = y_0, \qquad c_0 + c_1 x_1 + c_2 x_1^2 + c_3 x_1^3 = y_1 \tag{6.54}$$

Similarly, the second polynomial interpolates the data at the ends of its own subinterval:

$$d_0 + d_1 x_1 + d_2 x_1^2 + d_3 x_1^3 = y_1, \qquad d_0 + d_1 x_2 + d_2 x_2^2 + d_3 x_2^3 = y_2 \tag{6.55}$$

So far we have produced four equations, employing the definition of what it means to be an interpolating polynomial, $p(x_j) = y_j$. Obviously, that's not enough, so we'll have to use more properties. Specifically, we will impose the continuity of the first derivative at x_1:

$$c_1 + 2c_2 x_1 + 3c_3 x_1^2 = d_1 + 2d_2 x_1 + 3d_3 x_1^2 \tag{6.56}$$

and, similarly, the continuity of the second derivative at x_1:

$$2c_2 + 6c_3 x_1 = 2d_2 + 6d_3 x_1 \tag{6.57}$$

At this point, we have six equations for eight unknowns. This is a pattern that will re-emerge in the general case below. There are several ways via which we can produce two more constraints; here and in what follows, we will choose to make the second derivative go to 0 at the endpoints of the initial interval. This gives rise to what is known as a *natural spline*.[12] We get two more equations by demanding that the second derivative is 0 at x_0 and at x_2:

$$2c_2 + 6c_3 x_0 = 0, \qquad 2d_2 + 6d_3 x_2 = 0 \tag{6.58}$$

where, obviously, only the first polynomial can be used at x_0 and only the second one at

[12] One of the problems investigates a different choice.

x_2. The 8 equations can be written in matrix form as follows:

$$\begin{pmatrix} 1 & x_0 & x_0^2 & x_0^3 & 0 & 0 & 0 & 0 \\ 1 & x_1 & x_1^2 & x_1^3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & x_1 & x_1^2 & x_1^3 \\ 0 & 0 & 0 & 0 & 1 & x_2 & x_2^2 & x_2^3 \\ 0 & 1 & 2x_1 & 3x_1^2 & 0 & -1 & -2x_1 & -3x_1^2 \\ 0 & 0 & 2 & 6x_1 & 0 & 0 & -2 & -6x_1 \\ 0 & 0 & 2 & 6x_0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 6x_2 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_1 \\ y_2 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{6.59}$$

where the only (minor) subtlety involved the fifth and sixth equations. This is analogous to, but quite distinct from, the matrix equation we were faced with in Eq. (6.17).[13]

To summarize, we employed three types of constraints: (a) interpolation conditions, (b) continuity conditions, and (c) natural-spline conditions. In one of the problems, you are asked to implement Eq. (6.59) programmatically; instead of following that avenue, we now turn to a more systematic approach, which also has the added benefit of involving a banded (tridiagonal) system of equations.

6.3.2 General Case

In the previous, explicit, example, we had three points and two panels. In the general case, we have n points (x_j for $j = 0, 1, \ldots, n-1$) and $n-1$ panels. Similarly, the fact that there are n points in total means there are $n-2$ interior points (i.e., excluding x_0 and x_{n-1}). Each panel gets its own spline, i.e., corresponds to four unknown parameters; since there are $n-1$ panels in total, we are faced with $4n-4$ undetermined parameters. We will find these using the following constraints:

- A given spline should interpolate the data at its left endpoint:

$$s_{k-1,k}(x_{k-1}) = y_{k-1}, \qquad k = 1, 2 \ldots, n-1 \tag{6.60}$$

  and at its right endpoint:

$$s_{k-1,k}(x_k) = y_k, \qquad k = 1, 2 \ldots, n-1 \tag{6.61}$$

  Together, these are $2n-2$ conditions.
- For each of the $n-2$ interior points, the first derivative of the two splines on either side should match:

$$s'_{k-1,k}(x_k) = s'_{k,k+1}(x_k), \qquad k = 1, 2, \ldots, n-2 \tag{6.62}$$

  where this k ends at $n-2$. We get another $n-2$ conditions this way.

[13] Incidentally, the coefficient matrix in Eq. (6.59) belongs to the class of nonsymmetric matrices. As advertised in chapter 4, these show up quite often in practice.

- For each of the $n-2$ interior points, the second derivative of the two splines on either side should match:

$$s''_{k-1,k}(x_k) = s''_{k,k+1}(x_k), \qquad k = 1, 2, \ldots, n-2 \tag{6.63}$$

  This provides us with another $n-2$ conditions.
- So far, we have $4n-6$ conditions. To find the missing two constraints, we decide to use a *natural spline*, i.e., make the second derivative go to 0 at x_0 and at x_{n-1}:

$$s''_{0,1}(x_0) = s''_{n-2,n-1}(x_{n-1}) = 0 \tag{6.64}$$

  This brings the total number of conditions up to $4n-4$, which is equal to the number of undetermined parameters.

Let's start writing these conditions out. Our strategy will be to express our cubic spline $s_{k-1,k}(x)$ in terms of the input data (x_k and y_k) as well as the second-derivative values at the nodes. We call the latter c_k's; our relation giving the continuity of the second derivative, Eq. (6.63), is then simply:

$$s''_{k-1,k}(x_k) = s''_{k,k+1}(x_k) = c_k, \qquad k = 1, 2, \ldots, n-2 \tag{6.65}$$

We don't actually know these yet. For a natural spline, we do know that:

$$c_0 = c_{n-1} = 0 \tag{6.66}$$

as per Eq. (6.64). You may wish to think of the c_k's as y''_k's, though we denote them c_k's because we will end up solving a linear system of equations for them, as in earlier sections. This linear system, in its turn, will give us the c_k's in terms of the input data (x_k and y_k) directly. To belabor the obvious, in other sections of the present chapter the c_k's are coefficients of a polynomial, but here they are second-derivative values.

Remember that we are dealing with a $s_{k-1,k}(x)$ in Eq. (6.51) which is cubic, so $s''_{k-1,k}(x)$ is a straight line; thus, if we know the values of the second derivative at the left and right endpoint of a given subinterval, c_{k-1} and c_k at x_{k-1} and x_k, we can apply our standard Lagrange-interpolation formula from Eq. (6.22). This is:

$$s''_{k-1,k}(x) = c_{k-1}\frac{x - x_k}{x_{k-1} - x_k} + c_k\frac{x - x_{k-1}}{x_k - x_{k-1}} \tag{6.67}$$

As promised, we are expressing the second derivative of the spline in terms of the c_k's, namely the values of the second derivative at the nodes. The notation here can get a bit confusing, so make sure you remember the essential point: we are applying Lagrange interpolation for a straight line in $x_{k-1} \le x \le x_k$; just like in Eq. (6.51), we will eventually take $k = 1, 2 \ldots, n-1$.

If we now integrate Eq. (6.67) we can get an expression for the first derivative of the spline (at any x):

$$s'_{k-1,k}(x) = c_{k-1}\frac{x^2/2 - xx_k}{x_{k-1} - x_k} + c_k\frac{x^2/2 - xx_{k-1}}{x_k - x_{k-1}} + A \tag{6.68}$$

where we called the integration constant A. If we integrate another time, we can get the spline itself:

$$s_{k-1,k}(x) = c_{k-1}\frac{x^3/6 - x^2 x_k/2}{x_{k-1} - x_k} + c_k\frac{x^3/6 - x^2 x_{k-1}/2}{x_k - x_{k-1}} + Ax + B \tag{6.69}$$

where there is now another integration constant, B. If you use Eq. (6.69) twice to impose the conditions that the spline should interpolate the data at its left and right endpoints, Eq. (6.60) and Eq. (6.61), you can eliminate A and B (as you'll verify in a problem). Thus, your spline is now written only in terms of the input data (x_k and y_k) and the second-derivative values at the nodes (c_k). In equation form, this is:

$$\begin{aligned} s_{k-1,k}(x) = {} & y_{k-1}\frac{x_k - x}{x_k - x_{k-1}} + y_k\frac{x - x_{k-1}}{x_k - x_{k-1}} \\ & - \frac{c_{k-1}}{6}\left[(x_k - x)(x_k - x_{k-1}) - \frac{(x_k - x)^3}{x_k - x_{k-1}}\right] \\ & - \frac{c_k}{6}\left[(x - x_{k-1})(x_k - x_{k-1}) - \frac{(x - x_{k-1})^3}{x_k - x_{k-1}}\right] \end{aligned} \tag{6.70}$$

where $k = 1, 2 \ldots, n-1$. This equation identically obeys the relation giving the continuity of the second derivative, Eq. (6.63) or Eq. (6.65).[14] It does *not* in general obey the relation giving the continuity of the first derivative, Eq. (6.62), so let's impose it explicitly and see what happens. First, differentiate Eq. (6.70) to get:

$$\begin{aligned} s'_{k-1,k}(x) = {} & \frac{y_k - y_{k-1}}{x_k - x_{k-1}} + \frac{c_{k-1}}{6(x_k - x_{k-1})}\left(-3x^2 + 6xx_k - 2x_k^2 - 2x_k x_{k-1} + x_{k-1}^2\right) \\ & + \frac{c_k}{6(x_k - x_{k-1})}\left(3x^2 - 6xx_{k-1} + 2x_{k-1}^2 + 2x_k x_{k-1} - x_k^2\right) \end{aligned} \tag{6.71}$$

This function can be immediately evaluated at x_k to give:

$$s'_{k-1,k}(x_k) = \frac{y_k - y_{k-1}}{x_k - x_{k-1}} + \frac{x_k - x_{k-1}}{6}(2c_k + c_{k-1}) \tag{6.72}$$

We can also take $k \to k+1$ in Eq. (6.71) and then evaluate at x_k to get:

$$s'_{k,k+1}(x_k) = \frac{y_{k+1} - y_k}{x_{k+1} - x_k} - \frac{x_{k+1} - x_k}{6}(c_{k+1} + 2c_k) \tag{6.73}$$

Equating the right-hand sides in the last two relations, as per Eq. (6.62), allows us to find an equation for the unknown parameters c_k:

$$(x_k - x_{k-1})c_{k-1} + 2(x_{k+1} - x_{k-1})c_k + (x_{k+1} - x_k)c_{k+1} = 6\left(\frac{y_{k+1} - y_k}{x_{k+1} - x_k} - \frac{y_k - y_{k-1}}{x_k - x_{k-1}}\right) \tag{6.74}$$

[14] If you're not seeing this, go back to Eq. (6.67) and check it there.

where $k = 1, 2 \ldots, n-2$. Remember, as per Eq. (6.66), $c_0 = c_{n-1} = 0$ holds. When going over our strategy, we promised that we would produce a linear system that can be solved to find the c_k's in terms of the input data (x_k and y_k) directly; we have now set this up.

You might feel more convinced after seeing this in matrix form. Recall, the second derivative values go from c_0, c_1, all the way up to c_{n-2} and c_{n-1}, but we are only solving Eq. (6.74) for c_1 through c_{n-2}. The situation is similar to what we saw when discretizing the action around Fig. 5.16. We have n data points but only $n-2$ variables to solve for. The system of equations can be expressed as:

$$\begin{pmatrix} 2(x_2 - x_0) & x_2 - x_1 & 0 & \ldots & 0 \\ x_2 - x_1 & 2(x_3 - x_1) & x_3 - x_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \ldots & x_{n-3} - x_{n-4} & 2(x_{n-2} - x_{n-4}) & x_{n-2} - x_{n-3} \\ 0 & \ldots & 0 & x_{n-2} - x_{n-3} & 2(x_{n-1} - x_{n-3}) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-3} \\ c_{n-2} \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-3} \\ b_{n-2} \end{pmatrix} \tag{6.75}$$

where the b_k's are defined as per the right-hand side of Eq. (6.74): this makes the equation shorter, but also helps us think about how to implement things programatically below.

Our coefficient matrix here is symmetric tridiagonal (and also diagonally dominant); this means one could use very efficient solvers, but we'll go with our standard approach, Gaussian elimination with pivoting, despite it being overkill. Once we've solved the system in Eq. (6.75), we will have determined all the c_k's in terms of the input data. Then, we can use them in Eq. (6.70) to find the value of the interpolant at any x. There's actually a slightly subtle point involved here: Eq. (6.70) holds for a specific subinterval, namely $x_{k-1} \leq x \leq x_k$. When trying to find the value of the interpolant at x, we don't know ahead of time which subinterval x belongs to, so we'll also have to first determine that.

6.3.3 Implementation

Code 6.2 starts by importing the underlying $f(x)$, Runge's function from Eq. (6.18), as well as the function that produced a table of input data, (x_j, y_j), both from `barycentric.py`. We also import our Gaussian elimination function, which we'll need to determine the c_k's.

As before, the actual interpolation procedure will be carried out in two separate steps. First, we define a function that computes the values of the second derivative at the nodes, namely the c_k's, as per Eq. (6.74) or Eq. (6.75). This knows nothing about the specific x where we wish to interpolate: it only cares about the input data, (x_j, y_j). Since we have n data pairs, the cofficient matrix will have dimensions $(n-2)\times(n-2)$; similarly, the solution vector will be $(n-2) \times 1$, since we already know that $c_0 = c_{n-1} = 0$. We need to fill up a tridiagonal matrix, similarly to what we saw in section 5.6 when extremizing the action. We treat the main and other diagonals differently via slicing the arrays we pass as the first argument to `numpy.fill_diagonal()`, in addition to employing slicing to determine what makes up each diagonal. Setting up the right-hand side of the linear system is largely straightforward; once again `numpy`'s indexing makes things very convient: `dataxs[1:-1]` is easier to read than `dataxs[1:n-1]`. Slicing is used yet again to ensure that all the c_j's

splines.py Code 6.2

```
from barycentric import f, generatedata
from gauelim_pivot import gauelim_pivot
import numpy as np

def computecs(dataxs,datays):
     n = dataxs.size
     A = np.zeros((n-2,n-2))
     np.fill_diagonal(A, 2*(dataxs[2:]-dataxs[:-2]))
     np.fill_diagonal(A[1:,:], dataxs[2:-1]-dataxs[1:-2])
     np.fill_diagonal(A[:,1:], dataxs[2:-1]-dataxs[1:-2])
     b1 = (datays[2:]-datays[1:-1])/(dataxs[2:]-dataxs[1:-1])
     b2 = (datays[1:-1]-datays[:-2])/(dataxs[1:-1]-dataxs[:-2])
     bs = 6*(b1 - b2)

     cs = np.zeros(n)
     cs[1:-1] = gauelim_pivot(A, bs)
     return cs

def splineinterp(dataxs,datays,cs,x):
     k = np.argmax(dataxs>x)
     xk = dataxs[k]; xk1 = dataxs[k-1]
     yk = datays[k]; yk1 = datays[k-1]
     ck = cs[k]; ck1 = cs[k-1]

     val = yk1*(xk-x)/(xk-xk1) + yk*(x-xk1)/(xk-xk1)
     val -= ck1*((xk-x)*(xk-xk1) - (xk-x)**3/(xk-xk1))/6
     val -= ck*((x-xk1)*(xk-xk1) - (x-xk1)**3/(xk-xk1))/6
     return val

if __name__ == '__main__':
     dataxs, datays = generatedata(15, f, "equi")
     cs = computecs(dataxs, datays)
     x = 0.95; pofx = splineinterp(dataxs, datays, cs, x)
     print(x, pofx, f(x))
```

are stored in a single array: the solution of the linear system is an $(n-2)\times 1$ vector, so it's padded with $c_0 = c_{n-1} = 0$ to produce an $n \times 1$ vector.

With the table (x_j, y_j) and the c_j values in place, we turn to the second stage of the

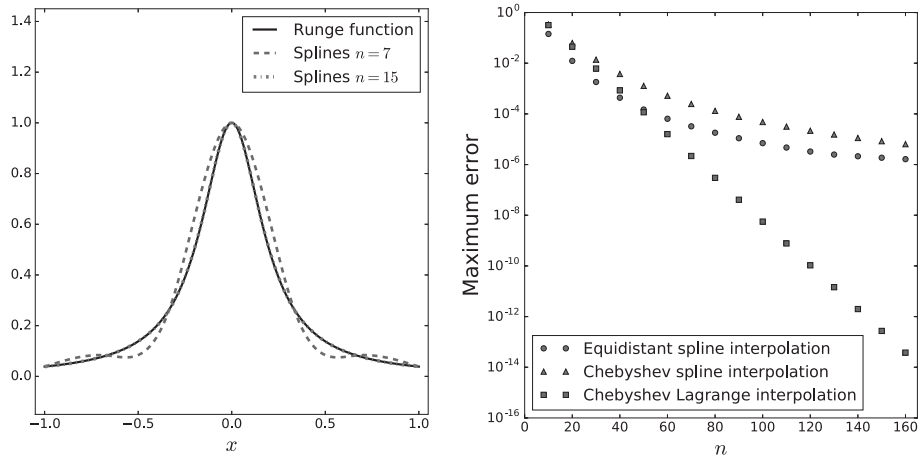


Fig. 6.7 Cubic-spline interpolation for equidistant nodes (left), vs other approaches (right)

interpolation process, namely the use of Eq. (6.70) in order to produce $p(x)$ at a given x. The main difficulty is determining which panel x falls in. We do so by using `numpy.argmax()`, which we also encountered in our implementation of Gaussian elimination with pivoting; as you may recall from our introduction to `numpy`, `numpy.argmax()` allows us to return the index of the maximum value. You should experiment to make sure you understand what's going on here: `dataxs>x` returns an array of boolean values (`True` or `False`). Since our x_j's are ordered, `dataxs>x` will give `False` for the x_j's that are smaller than x and `True` for the x_j's that are larger than x. Then, `numpy.argmax()` realizes that `True > False`, so it returns as `k` the index for the first x_j that is larger than x. We are fortunate that if the maximum value appears more than once, `numpy.argmax()` returns the first occurrence (all the other x_j's to the right of x_k are also larger than x). The rest of the function is near-trivial: Eq. (6.70) is algebra-heavy, so we first define some local variables to make our subsequent code match the mathematical expressions as closely as possible.

The main program produces the data, constructs the c_j's, and then evaluates the interpolating polynomial $p(x)$. In order to highlight that cubic-spline interpolation can handle equally spaced grids well, we produce equidistant grid points and also pass in $x = 0.95$, a value for which Lagrange interpolation (at equidistant points) was doing quite poorly in the right panel of Fig. 6.6. The output of running this code is:

```
0.95 0.0426343358892 0.042440318302387266
```

The match is very good, despite the fact that we only used 15 points. As before, we decide to be more thorough, using this code at thousands of points to produce a quasi-continuous curve. Once again, we'll use a reasonably small number of nodes ($n = 7$ or $n = 15$), but thousands of x's.

The result is shown in the left panel of Fig. 6.7; as we did earlier, we are not showing the data points (x_j, y_j) so you can focus on the curves. Note that we are only showing the case

of equidistant points, where the splines are doing a better job (but see below for splines with Chebyshev nodes). Overall, both $n = 7$ and $n = 15$ do a better job than Lagrange interpolation did in Fig. 6.6. This bears repeating: cubic-spline interpolation does just fine when you use an equally spaced grid, in contradistinction to Lagrange interpolation. As a matter of fact, the $n = 15$ results are virtually indistinguishable from the underlying Runge function. While there were no singularities to be dealt with in this example, in practice it is very common to employ cubic-spline interpolation with an equidistant grid starting near the singularity, given little other knowledge about the behavior of the underlying function.

Our finding in the left panel of Fig. 6.7 may be a bit surprising given our earlier admiration for the barycentric formula at Chebyshev nodes. We decide to investigate things further, still for Runge's function using cubic-spline interpolation, only this time we will keep increasing n and also explore both equidistant and Chebyshev nodes. While we're at it, we will also repeat this exercise for Lagrange interpolation with Chebyshev nodes (we already know that equidistant nodes are bad for Lagrange interpolation, so we don't use them). Since we wish to plot the result for several n's, we must somehow quantify how well a given interpolant $p(x)$ matches $f(x)$; inspired by the *infinity norm*, see Eq. (4.37), we will calculate the maximum magnitude of the difference $p(x) - f(x)$ across all the x's we are interpolating at. The result for such a calculation with all three methods is shown in the right panel of Fig. 6.7. First, observe that spline interpolation with equidistant nodes consistently outperforms spline interpolation with Chebyshev nodes (by a little). Second, comparing spline interpolation with equidistant nodes to Lagrange interpolation with Chebyshev nodes we see that for small n the splines do a better job, as we also saw in the left panel. However, as n keeps increasing, the splines "level off", i.e., do as well as they're going to at a maximum error of $\approx 10^{-6}$. In contradistinction to this, the Lagrange-interpolation results keep getting better, reaching close to machine precision.

The results of the right panel of Fig. 6.7 are significant and bear emphasizing: a single polynomial of degree a couple of hundred can describe Runge's function to within machine precision, if you're picking the nodes at Chebyshev points. This is a linear trend on a semilog plot: as you add 10 more points, you improve by an order of magnitude! Perhaps you can now appreciate the comments we made after deriving our general error formula for polynomial interpolation, Eq. (6.38): for analytic functions Lagrange interpolation at Chebyshev nodes converges geometrically, so it's likely the tool you are looking for.

6.4 Trigonometric Interpolation

We now turn to a related topic: what do you do if the data points you are faced with are *periodic*? In this case, obviously, your interpolatory function should also be periodic, so using a polynomial (as in previous sections) is not on the face of it appropriate.[15] In practice, one employs an expansion in sines and cosines. The notation, as well as the concepts,

[15] Though, even here, the barycentric formula with Chebyshev nodes does a great job, as you'll see in a problem. If you squint, you can see that polynomial interpolation at Chebyshev points is actually equivalent to trigonometric interpolation; one picks the x_j's trigonometrically and the other does the same for the ϕ_k's.

involved in this context can get a bit confusing, so let's try to take things from the start. We will use a number of tools that have the name of Fourier attached to them but, since they serve different purposes, it's good to begin at a very basic level. This way (it is hoped) you will start getting comfortable with our symbols and their use, while things are still reasonably simple.

6.4.1 Fourier Series

Assume we are dealing with a function $f(x)$ that is periodic; for simplicity, let us focus on the interval $[0, 2\pi]$.[16] If $f(x)$ is reasonably continuous, then a standard result is that it can be decomposed in the following *Fourier series*:

$$f(x) = \frac{1}{2}a_0 + \sum_{k=1}^{\infty} (a_k \cos kx + b_k \sin kx) \tag{6.76}$$

Note that the sum here extends from 1 to ∞. We call the k's "wave numbers", or sometimes even "frequencies", a left-over from the use of Fourier analysis that relates time, t, to frequency, ω or f. Note also that the zero-frequency term a_0 has been singled out and treated differently (with a 2 in the denominator), for reasons that will soon become clear.[17]

Crucially, for well-behaved functions $f(x)$, this decomposition is *exact*, in the sense that the series on the right-hand side converges to $f(x)$; thus, a general periodic $f(x)$ is re-written as a sum of sines and cosines. We can imagine a careful student, who solved the relevant problem in chapter 2, protesting at this point: what about the Gibbs phenomenon? As you may recall, this is a "ringing" effect which appears when you try to reconstruct a discontinuous function using a Fourier series; even as you take the number of terms in the series to infinity, this mismatch between the left-hand side and the right-hand side does not go away. In what follows, we will assume that our functions are sufficiently smooth so that such problems do not arise.

We can exploit the *orthogonality* of sines and cosines to extract the *Fourier coefficients* in Eq. (6.76), namely the a_k's and b_k's. As you will show in a problem, multiplying Eq. (6.76) with $\cos jx$ (and, separately, with $\sin jx$) and integrating x from 0 to 2π we get:

$$a_k = \frac{1}{\pi}\int_0^{2\pi} dx f(x)\cos kx, \quad k = 0, 1, \ldots, \qquad b_k = \frac{1}{\pi}\int_0^{2\pi} dx f(x)\sin kx, \quad k = 1, 2, \ldots \tag{6.77}$$

In both of these equations we renamed $j \to k$ after we were done. You can now see why we included a 2 in the denominator for the case of a_0 in Eq. (6.76): it leads to the same expression, Eq. (6.77), for both $k = 0$ and $k > 0$.

In what follows, we will benefit from employing the Fourier series in a different form.

[16] You can trivially scale this to $[0, T]$ later on, by writing $x = 2\pi t/T$.
[17] We could have also included a b_0 term, but it wouldn't have made a difference.

First, recall *Euler's formula* (in its incarnation as *de Moivre's formula*):

$$e^{ikx} = \cos kx + i \sin kx \tag{6.78}$$

where $i = \sqrt{-1}$ is the *imaginary unit*. Making (repeated) use of this formula, the Fourier series can be re-written as:

$$f(x) = \sum_{k=-\infty}^{\infty} c_k e^{ikx} \tag{6.79}$$

as you will show in a problem. It's important to notice that the k here goes from $-\infty$ to $+\infty$. In another problem you will also show how to solve for the coefficients c_k:

$$c_k = \frac{1}{2\pi} \int_0^{2\pi} dx f(x) e^{-ikx}, \qquad \text{integer } k \tag{6.80}$$

where this time we employed the orthogonality of the plane waves. Incidentally, this is one of the few points in the book where we are choosing to employ complex numbers.

Before we conclude this section, let us observe that there is an asymmetry at play here: we employ a sum to decompose our function $f(x)$, see Eq. (6.76) or Eq. (6.79), but we use integrals to get the coefficients a_k, b_k, or c_k in terms of the function $f(x)$, see Eq. (6.77), or Eq. (6.80).

6.4.2 Finite Series: Trigonometric Interpolation

In the present section, we take the more practical route of assuming a set of n data pairs are known, and trying to reconstruct those using ideas borrowed from Fourier analysis. This approach is known as *trigonometric interpolation* for good reason. Since we're dealing with a finite number of data points, we will employ a series with a finite number of terms and make sure that no integrals appear at any point in our calculation.

Definition and Properties

We start from the same assumption as in earlier sections on interpolation: we are given n data points (also known as *support points*) where we know both the x and the y value: (x_j, y_j) for $j = 0, 1, \ldots, n-1$ using our standard Python-inspired 0-indexing.[18] As was also the case earlier in the chapter, we don't know if these points truly come from a given analytic function $f(x)$ (in which case $f(x_j) = y_j$), or were experimentally measured, and so on. All we know is that we need to produce an interpolating function, which *must* go through the points we are given, in addition to satisfying criteria relating to smoothness etc. Essentially, the only difference from earlier on is that our problem now involves x_j's which lie in the interval $[0, 2\pi]$ and y_j's which are periodic; thus, whatever interpolatory function we come up with had better be periodic as well.

[18] We will be interested only in the case where the y_j are real numbers.

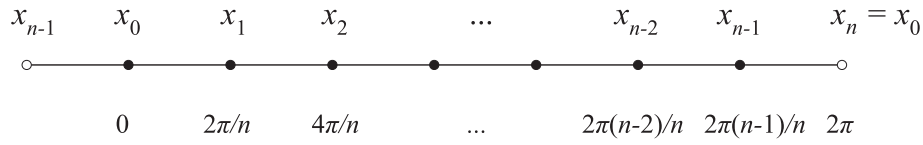


Fig. 6.8 Grid for the x_j's, with solid dots showing our actual points and open dots implied ones

Since we know only the value of y_j at n distinct values of x_j, it is unreasonable to expect that we will be able to determine infinitely many a_k and b_k coefficients from the integrals of Eq. (6.77). Instead, what we do is to truncate the series in Eq. (6.76); since we have n data points, the corresponding interpolating polynomial will be:[19]

$$p(x) = \frac{1}{2}a_0 + \sum_{k=1}^{m} (a_k \cos kx + b_k \sin kx) \tag{6.81}$$

where we've implicitly assumed that we are dealing with the case of n-odd. To see why this is so, count the total number of undetermined parameters; these are $1+m+m = 2m+1$ which means that we'll be able to determine them by matching at our n data points, $p(x_j) = y_j$, if $n = 2m + 1$. Observe that we haven't assumed that the x_j are equally spaced so far.

For reasons that will later become clear, we prefer to focus on the case of n-even, so we cannot use Eq. (6.81). Instead, our main formula for trigonometric interpolation will be:

$$p(x) = \frac{1}{2}a_0 + \sum_{k=1}^{m-1} (a_k \cos kx + b_k \sin kx) + \frac{1}{2}a_m \cos mx \tag{6.82}$$

This time the total number of parameters is $1 + 2(m - 1) + 1 = 2m$; thus, we will be able to determine all of them by matching via $p(x_j) = y_j$, if $n = 2m$. We see that the first and last terms are treated separately. Since n (or m) is finite, this $p(x)$ will be an approximation to an (underlying) "true function" $f(x)$. A complementary viewpoint is that $f(x)$ is not part of our problem here: we are simply faced with a table of input data points, (x_j, y_j) for $j = 0, 1, \ldots, n - 1$, and we are doing our best to produce a function, $p(x)$, which goes through those points and also is reasonably behaved everywhere else. Crucially, we will be interested in the case where both x_j and y_j are real; then, our $p(x)$ will also be real-valued.

Observe that applying Eq. (6.82) n times leads to an $n\times n$ linear system of equations, with $n = 2m$ unknowns. If there were no special tricks we could apply, solving this system would require $O(n^3)$ operations, just as we observed in our general comments in section 6.1.2.

While nothing so far assumes anything specific about the placement of the x_j values, it is very convenient to study the special case of equidistant values. Since we are dealing with the interval $[0, 2\pi]$ and n points in total, this naturally leads to a grid:

[19] It's not yet clear why this is called a *polynomial*, but bear with us.

$$x_j = \frac{2\pi j}{n}, \qquad j = 0, 1, \ldots, n-1 \tag{6.83}$$

As always, we are placing n points. While the value 0 is included in our x_j's (it's x_0), the value 2π isn't; this is because we know our signal is periodic, so if we had included an x_j at 2π then its corresponding y_j value would have been identical to y_0. Instead of storing needless information, we simply stop our points just before we get to 2π. This is illustrated in Fig. 6.8, where we are showing both the j index and the x_j value.

If we want to solve for the a_k and b_k parameters, the avenue that led to, say, Eq. (6.77) is now closed to us: we can't integrate Eq. (6.82) from 0 to 2π because we don't actually know $p(x)$; that is precisely what we are trying to compute. All we have at our disposal are the $p(x_j) = y_j$ values, available only at the grid points. With that in mind, we will, instead, employ the pleasant fact that sines and cosines are orthogonal to each other *even in the discrete case* for our equally spaced grid! In other words:

$$\begin{aligned}
&\sum_{j=0}^{n-1} \cos kx_j \cos lx_j = \begin{cases} 0, & k \neq l \\ m, & 0 < k = l < m \\ 2m, & k = l = 0 \text{ or } k = l = m \end{cases} \\
&\sum_{j=0}^{n-1} \cos kx_j \sin lx_j = 0 \\
&\sum_{j=0}^{n-1} \sin kx_j \sin lx_j = \begin{cases} 0, & k \neq l \\ m, & 0 < k = l < m \\ 0, & k = l = 0 \text{ or } k = l = m \end{cases}
\end{aligned} \tag{6.84}$$

You are guided toward these remarkable properties in a problem. Armed with Eq. (6.84), we can immediately solve for our a_k and b_k parameters:

$$\begin{aligned}
a_k &= \frac{1}{m} \sum_{j=0}^{n-1} y_j \cos kx_j, \qquad k = 0, 1, \ldots, m \\
b_k &= \frac{1}{m} \sum_{j=0}^{n-1} y_j \sin kx_j, \qquad k = 1, 2, \ldots, m-1
\end{aligned} \tag{6.85}$$

Both of these are given in terms of y_j, the known input data. Pay close attention to the values k can take on in each case; these are fully consistent with what we need in Eq. (6.82). Speaking of which, we are now all set: we get our parameters from Eq. (6.85) and our interpolating polynomial $p(x)$ at any value x from Eq. (6.82); obviously, if we did things correctly, $p(x)$ should match the input data at the grid points, i.e., $p(x_j) = y_j$.

Note that, having employed the discrete orthogonality, our problem of Eq. (6.85) requires only $O(n^2)$ operations to implement: we need $O(n)$ operations for a given k and are

dealing with n values of k in total. This is already a major improvement (n^2 vs n^3); even so, we will soon find out that we can do even better than that.

Implementation

We remember that we dealt with a periodic function all the way back in chapter 3; this was $f(x) = e^{\sin(2x)}$, though the periodicity didn't play a role at the time. As in the case of polynomial interpolation earlier in this chapter, we first *produce* our data starting from our chosen function, followed by interpolating and comparing. Code 6.3 starts by defining a function to represent $f(x) = e^{\sin(2x)}$: observe that this employs `numpy` functionality; this shouldn't matter if we're passing in a single number but, as we'll soon discover, we can also use this function with a whole `numpy` array as an argument and it works equally seamlessly. We then define a function that picks the grid of x_j values as per Eq. (6.83) and produces the y_j values as $y_j = f(x_j)$; after this, $f(x)$ is no longer needed.

With the table of data (x_j, y_j) in place, we proceed to tackle the problem of trigonometric interpolation in two steps. We first create a function that evaluates our a_k and b_k parameters, as per Eq. (6.85). It is important to do this separately, since we don't want to re-evaluate these parameters each time we pick a different x at which to interpolate. The function `computeparams()` first sets up the dimensions of the two arrays, closely following Eq. (6.85); the values k can take for a_k and for b_k are different, so that is also reflected in the loops that evaluate our parameters. Note that, regardless of which of the two equations in Eq. (6.85) we are dealing with, the right-hand side always involves n terms in the sum; as a result, we have implemented both right-hand sides as `@` products of one-dimensional arrays, thereby obviating the need for a second loop and a `j` index. When storing the results in arrays, we have to be a bit careful in the case of `bparams`, since $k = 1, 2, \ldots, m-1$ but `numpy` array indices start at 0; thus, all elements are shifted down by one (but not on the right-hand side, where Eq. (6.85) involves the actual k value).

The second step in trigonometric interpolation, now that the a_k and b_k parameters are in place, is to use Eq. (6.82) to compute the value of the interpolating polynomial $p(x)$ at a given x. In the function `triginterp()` we first take care of the a_0 a_m, which need to be handled separately. We then step through the terms in the sum, handling $k = 1, 2, \ldots, m-1$ one at a time. Note that we could have employed `@` multiplications here instead of a loop, as long as we were careful in treating the `bparams` appropriately. This would have involved a new array `ks = np.arange(1,m)` and slicing.

The main program simply calls three of the functions we defined. As before, we use `dataxs` for the input data x_j's and `x` for the x where we wish to interpolate; the value of the latter is picked at random. In Fig. 6.9 we are showing the $f(x)$ we started from. More importantly, we show the data (x_j, y_j) together with the result of our interpolating polynomial at many values of x. The left panel is for $n = 6$: the general trend is roughly captured, despite using so few points.[20] The $n = 8$ case (right panel) is even better. As you will find out when you experiment for yourself, from 14 points and up you cannot tell the two curves apart with the naked eye.

[20] You should try using $n = 4$ points. Did you see the result coming?

triginterp.py Code 6.3

```
from math import pi
import numpy as np

def f(x):
     return np.exp(np.sin(2*x))

def generatedata(n,f):
     dataxs = 2*pi*np.arange(n)/n
     datays = f(dataxs)
     return dataxs, datays

def computeparams(dataxs,datays):
     n = dataxs.size
     m = n//2
     aparams = np.zeros(m+1)
     bparams = np.zeros(m-1)

     for k in range(m+1):
          aparams[k] = datays@np.cos(k*dataxs)/m
     for k in range(1,m):
          bparams[k-1] = datays@np.sin(k*dataxs)/m
     return aparams, bparams

def triginterp(aparams,bparams,x):
     n = aparams.size + bparams.size
     m = n//2
     val = 0.5*(aparams[0] + aparams[-1]*np.cos(m*x))
     for k in range(1,m):
          val += aparams[k]*np.cos(k*x)
          val += bparams[k-1]*np.sin(k*x)
     return val

if __name__ == '__main__':
     dataxs, datays = generatedata(6, f)
     aparams, bparams = computeparams(dataxs, datays)
     x = 0.3; pofx = triginterp(aparams, bparams, x)
     print(x,pofx)
```

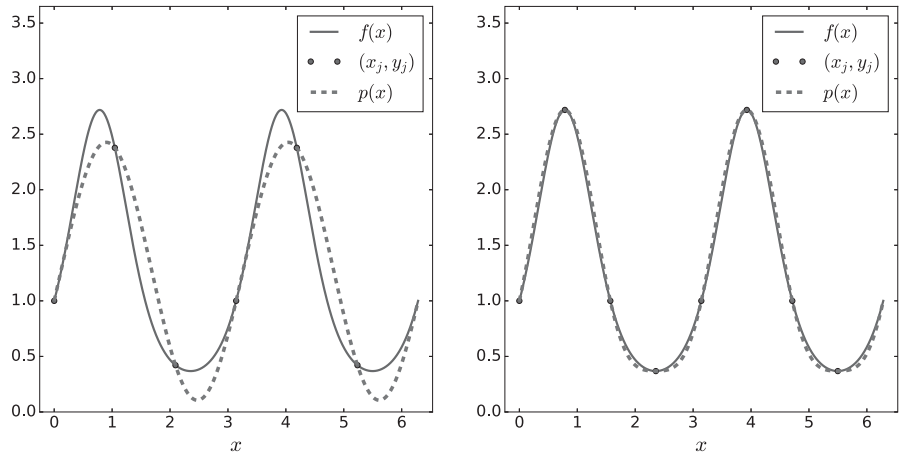


Fig. 6.9 Trigonometric interpolation for the cases of $n = 6$ (left) and $n = 8$ (right)

Keep in mind that our equations and code above only work for the case of $n = 2m$, i.e., even-n. In a problem, you are asked to extend the formalism to the case of odd-n but our main line of development will assume that n is even; as a matter of fact, below we will be yet more specific, requiring n to be a power of 2.

6.4.3 Discrete Fourier Transform

We could decide to stop here: combined with our earlier sections on Lagrange interpolation and cubic splines, we already have a toolbox that is sufficiently varied to handle a large number of approximation problems. However, the connections of trigonometric interpolation to the wider theme of Fourier transforms are too good to ignore; this will also allow us to introduce one of the most successful algorithms ever, the *fast Fourier transform* (FFT). Thus, we will now reformulate our earlier work with sines and cosines to use complex exponentials. However, we will not be doing this just for the sake of abstractly introducing the FFT, but will also see how to use this new algorithm to carry out interpolation for periodic problems. It's important to keep in mind that the literature on this subject is riddled with errors; we therefore choose to explicitly derive our interpolation formula step-by-step.

First Definition

When faced with the infinite Fourier series, we saw that this could be given in either a real form (sines and cosines of Eq. (6.76)) or in a complex form (complex exponentials of Eq. (6.79)). In the former case our sum was over positive k's and in the latter over both positive and negative k's. When we moved to the real finite series, Eq. (6.82), we summed over positive k's and were careful in handling the extra cosine that arose for even-n. We will now make an analogous transition from sines and cosines to complex exponentials for the finite series, again for even-n (i.e., we are still taking $n = 2m$):

$$
\begin{aligned}
p(x) &= \frac{1}{2}a_0 + \sum_{k=1}^{m-1}(a_k \cos kx + b_k \sin kx) + \frac{1}{2}a_m \cos mx \\
&= \frac{1}{2}a_0 + \sum_{k=1}^{m-1}\left(a_k \frac{e^{ikx}+e^{-ikx}}{2} + b_k \frac{e^{ikx}-e^{-ikx}}{2i}\right) + \frac{1}{2}a_m \cos mx \\
&= \frac{1}{2}a_0 + \sum_{k=1}^{m-1}\frac{1}{2}(a_k - ib_k)e^{ikx} + \sum_{k=1}^{m-1}\frac{1}{2}(a_k + ib_k)e^{-ikx} + \frac{1}{2}a_m \cos mx \\
&= \frac{1}{2}a_0 + \sum_{k=1}^{m-1}\frac{1}{2}(a_k - ib_k)e^{ikx} + \sum_{k=-m+1}^{-1}\frac{1}{2}(a_{-k} + ib_{-k})e^{ikx} + \frac{1}{2}a_m \cos mx \\
&= c_0 + \sum_{k=1}^{m-1} c_k e^{ikx} + \sum_{k=-m+1}^{-1} c_k e^{ikx} + c_{-m} \cos mx
\end{aligned}
\tag{6.86}
$$

In the first line we simply wrote down our interpolating polynomial from Eq. (6.82). In the second line we used de Moivre's formula (a couple of times) from Eq. (6.78). In the third line we grouped terms into two sums. In the fourth line we took $k \to -k$ in the second sum, also appropriately adjusting the values k can take on. In the fifth line we introduced a new set of c_k parameters, which are related to the a_k and b_k parameters in the following way:

$$
\begin{aligned}
c_0 &= \frac{1}{2}a_0, \qquad & c_k &= \frac{1}{2}(a_k - ib_k), \qquad & k &= 1, 2, \ldots, m-1 \\
c_{-m} &= \frac{1}{2}a_m, \qquad & c_k &= \frac{1}{2}(a_{-k} + ib_{-k}), \qquad & k &= -m+1, -m+2, \ldots, -2, -1
\end{aligned}
\tag{6.87}
$$

The definition of c_{-m} was a bit arbitrary, since we could have just as well called this c_m (more on this below). We notice that the first three terms on the last line of Eq. (6.86) can be grouped into a single sum:

$$
p(x) = \sum_{k=-m+1}^{m-1} c_k e^{ikx} + c_{-m} \cos mx \tag{6.88}
$$

Crucially, this goes over both positive and negative k values, similarly to the infinite-series case in Eq. (6.79).[21] In contradistinction to Eq. (6.79), here we do not have only complex exponentials: the c_{-m} is treated separately and gets a cosine.[22]

Recall that our problem started out with $n = 2m$ input data values y_j and Eq. (6.82) expressed $p(x)$ in terms of the a_k and b_k parameters: there were $n = 2m$ of these. In the present case of Eq. (6.88), counting up the c_k parameters, we find there are $2(m-1)+1+1 = 2m = n$ parameters, so everything is consistent. Note that we could have employed de Moivre's formula for the a_m term too in our derivation of Eq. (6.86); this would have led

[21] Incidentally, if you set $z = e^{ix}$, then the sum is over z^k: this is *polynomial interpolation on the unit circle*. Of course, k can also be negative, so this would be a *Laurent polynomial*.

[22] This would not have been the case if we were studying the odd-n problem.

to our sum over k going from $-m$ to m, all for complex exponentials. Even so, the special treatment of this first/last term would not have been avoided: we would have been faced with a c_m and a c_{-m} that would have been equal. While it's not so pretty to single out c_{-m} as in Eq. (6.88), it would have been even stranger to use $2m+1$ parameters c_k but have one of them be non-independent.[23]

Up to this point, our main result is the new version of the interpolating polynomial, Eq. (6.88). This was a polynomial, $p(x)$, that can be evaluated at any value of x we may choose. We now change gears, to address the more specific problem of using our grid of x_j values from Eq. (6.83). In other words, we will (once again) assume we are faced with a table of input data points, (x_j, y_j) for $j = 0, 1, \ldots, n-1$; we wish to compute the c_k's in terms of these known values. While we could produce an answer by using Eq. (6.85) to find the a_k and b_k parameters and then use those, in turn, via Eq. (6.87), to find the c_k parameters, we won't do that. Instead, we will try to work with Eq. (6.88) directly and attempt to find c_k in terms of x_j and y_j.

We start by evaluating the interpolating polynomial at the grid points:

$$p(x_j) = y_j = \sum_{k=-m+1}^{m-1} c_k e^{ikx_j} + c_{-m}\cos mx_j = \sum_{k=-m+1}^{m-1} c_k e^{ikx_j} + c_{-m}e^{-imx_j} = \sum_{k=-m}^{m-1} c_k e^{ikx_j} \tag{6.89}$$

The first equality simply assumed that the interpolation worked, i.e., we get the desired y_j values at x_j. The second equality just plugged in x_j to Eq. (6.88). The third equality made use of the evenness of cosines, $\cos(-mx_j) = \cos mx_j$, as well as the fact that $\sin mx_j$ vanishes at our grid points:

$$\sin(-mx_j) = -\sin mx_j = -\sin\left(m\frac{2\pi j}{2m}\right) = -\sin \pi j = 0 \tag{6.90}$$

As a result, we are free to pretend that we were faced with a complex exponential, e^{-imx_j}, instead of only a cosine, $\cos mx_j$. This allows us to modify our sum so that it runs from $-m$ to $m-1$; this time, we are justified in doing so, because we are limited to points on the grid. Our main result here is known as the *inverse discrete Fourier transform* (inverse DFT), for reasons that will soon become clear:

$$y_j = \sum_{k=-m}^{m-1} c_k e^{ikx_j}, \qquad j = 0, 1, \ldots, n-1 \tag{6.91}$$

It's important to realize that this equation relates y_j to c_k. In other words, it refers *only* to grid points as per Eq. (6.83); n numbers are transformed into n numbers. The problem, of course, is that we do not (yet) know the values of c_k; this is what we are trying to solve for.

We now wish to have a formula that goes in the opposite direction: starting from the values of x_j and y_j, compute c_k. We will accomplish this by combining Eq. (6.91) with

[23] There is also the option of having the sum in Eq. (6.88) go from $-m$ to $m-1$ by fiat; this leads to an interpolating polynomial with an imaginary part and is therefore different from what we were doing in Eq. (6.82).

a(nother) remarkable property, namely the fact that complex exponentials are orthogonal to each other *even in the discrete case*:

$$\sum_{j=0}^{n-1} e^{ikx_j} e^{-ilx_j} = \begin{cases} n, & (k-l)/n \text{ is integer} \\ 0, & \text{otherwise} \end{cases} \tag{6.92}$$

A problem guides you toward proving this. Knowing that sines and cosines are orthogonal to each other even for the discrete case, Eq. (6.84), this perhaps does not come as a total surprise. As advertised, we now multiply Eq. (6.91) with e^{-ilx_j} and sum over all the x_j's:

$$\sum_{j=0}^{n-1} e^{-ilx_j} y_j = \sum_{k=-m}^{m-1} c_k \sum_{j=0}^{n-1} e^{ikx_j} e^{-ilx_j} = \sum_{k=-m}^{m-1} c_k n \delta_{k,l} = n c_l \tag{6.93}$$

In the penultimate step we used the orthogonality property from Eq. (6.92), while acknowledging that our k and l values run only from $-m$ to $m-1$; thus, the only contribution comes from the case $k = l$. We now rewrite our result in Eq. (6.93) by renaming $l \to k$:

$$c_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j e^{-ikx_j}, \qquad k = -m, -m+1, \ldots, m-2, m-1 \tag{6.94}$$

This, our desired result, is known as the *discrete Fourier transform* (DFT). It produces all $2m$ parameters c_k, starting from our input points y_j. These were precisely the c_k parameters we needed in order to evaluate the interpolating polynomial from Eq. (6.88). Once again, there is nothing continuous going on in Eq. (6.94): n numbers are transformed into n numbers. Our result also explains why Eq. (6.91) is known as the *inverse DFT*, since that equation recovers the data points y_j once you know the parameters c_k.

Let's quickly summarize a few points before things get out of hand; this paragraph will be somewhat dense, but may help you in the future. Keep in mind that in the case of an *infinite Fourier series* we are faced with a sum (expanding the function in terms of the Fourier coefficients) and an integral (computing the Fourier coefficients in terms of the function). On the other hand, in the case of *trigonometric interpolation* or the *discrete Fourier transform* we have a sum (direct) and a sum (inverse); interpolation deals with any x value whereas the DFT deals only with grid points x_j. Note that both the infinite Fourier series and the discrete Fourier transform apply to the case of a periodic function/signal. If you are faced with a nonperiodic function, you can simply focus on a given interval (assuming the function *does* repeat outside) and try to decribe only that region; your predictions will only be applicable to the interval you focused on (since, in reality, the function *doesn't* repeat outside the region of interest).[24] Incidentally, nonperiodic functions give rise to the *(continuous) Fourier transform*, which involves an integral (direct) and an integral (inverse). In this book we have little to say on continuous Fourier transforms, since we are always discretizing. While we arrived at the discrete Fourier transform starting from trigonometric interpolation, you will not be surprised to hear that you can also view the DFT as a discretization of the continuous Fourier transform.

[24] A more detailed study would also address topics like aliasing, windowing, and leakage.

Shifted Version

We will now slightly tweak our results. This is done both in anticipation of introducing the fast Fourier transform in the following subsection, and also in order to make contact with notation that is standard on this subject. We will make three related changes.

First, we observe that our expressions in Eq. (6.94) and in Eq. (6.91) involve x_j, which are our grid points from Eq. (6.83). Let's use that equation to re-express the DFT:

$$c_k = \frac{1}{n}\sum_{j=0}^{n-1} y_j e^{-2\pi i k j/n}, \qquad k = -m, -m+1, \ldots, m-2, m-1 \tag{6.95}$$

This ikj in the exponent may be confusing, so keep in mind that i is the imaginary unit, k is our wave number (or "frequency") index, and j is our spatial index. Second, we note that our expressions in Eq. (6.94) and in Eq. (6.91) exhibit an asymmetry: j is non-negative, going from 0 to $n-1$, whereas k can be either negative or positive, going from $-m$ to $m-1$. As it turns out, this asymmetry can be lifted; to do so, observe that Eq. (6.95) can help us show that the DFT is periodic in k, with period n:

$$c_{k+n} = \frac{1}{n}\sum_{j=0}^{n-1} y_j e^{-2\pi i (k+n) j/n} = \frac{1}{n}\sum_{j=0}^{n-1} y_j e^{-2\pi i k j/n} e^{-2\pi i j} = c_k \tag{6.96}$$

since $e^{-2\pi i j} = 1$. Given that up to this point our k could also take on negative values, we can use $c_{k+n} = c_k$ to re-express c_k for the negative frequencies $k = -m, -m+1, \ldots, -1$ in terms of positive frequencies $k = m, m+1, \ldots, n-1$.[25] Third, it is customary to have the $1/n$ in the definition of the *inverse* DFT, not in that of the DFT, cf. Eq. (6.91) and Eq. (6.94).

Putting these three modifications together, we arrive at a new definition of the DFT. In order to keep our notation straight, we will use a new symbol for the new version of our Fourier parameters, $\tilde{y}_k$, where k now goes from 0 to $n-1$:

$$\tilde{y}_k = \sum_{j=0}^{n-1} y_j e^{-2\pi i k j/n}, \qquad k = 0, 1 \ldots, n-1 \tag{6.97}$$

As before, this is simply transforming n numbers into n numbers. Observe how there is no x_j left over here; the assumption that we have equally spaced points has already been used, so our expressions from now on will only involve y_j and $\tilde{y}_k$. Crucially, both our j and our k indices now run from 0 to $n-1$.

If you're having trouble seeing the correspondence between Eq. (6.94) and Eq. (6.97), keep in mind that (a) we've replaced x_j with j as per Eq. (6.83), (b) we've shifted the negative frequencies so that they appear after the positive frequencies, and (c) there is no $1/n$ in the new definition of the DFT. The last two modifications can be summarized as follows:

$$\begin{pmatrix} \tilde{y}_0 & \tilde{y}_1 & \ldots & \tilde{y}_{m-1} \end{pmatrix}^T = n \begin{pmatrix} c_0 & c_1 & \ldots & c_{m-1} \end{pmatrix}^T$$

[25] Observe that those slots were *not* previously taken: our positive frequencies were only going up to $m-1$.

$$\begin{pmatrix} \tilde{y}_m & \tilde{y}_{m+1} & \dots & \tilde{y}_{n-1} \end{pmatrix}^T = n \begin{pmatrix} c_{-m} & c_{-m+1} & \dots & c_{-1} \end{pmatrix}^T \tag{6.98}$$

In addition to the (arbitrary) choice to remove the $1/n$, this also shows that the (zero and) positive frequencies in c_k are still in the same place when using $\tilde{y}_k$, whereas the negative frequencies in c_k are now placed in the "second half" of the available frequency slots of $\tilde{y}_k$. Keep in mind that the c_{-m} can be viewed as corresponding to either the largest-magnitude negative frequency or the largest-magnitude positive frequency: $c_{-m} = c_m$, as a consequence of $c_{k+n} = c_k$. Note that the 0th-frequency component, $\tilde{y}_0$, was also the 0th-frequency component of c_k; it's sometimes called the DC component: you can see from Eq. (6.97) that it is simply the sum of all the y_j's. Overall, we have now switched to what is known as the "standard order" of the DFT; we will sometimes refer to it as the "shifted version", as in the current section's heading.

We can also use our new Fourier parameters $\tilde{y}_k$ to rewrite our definition of the *inverse DFT* from Eq. (6.91). This has now turned into:

$$y_j = \frac{1}{n} \sum_{k=0}^{n-1} \tilde{y}_k e^{2\pi i k j/n}, \qquad j = 0, 1 \dots, n-1 \tag{6.99}$$

To see that this is true, you could start from Eq. (6.97) and employ the complex discrete orthogonality of Eq. (6.92). As in the case of the updated (direct) DFT, the x_j's are gone and the k's are now non-negative. Furthermore, there is now a $1/n$ term on the right-hand side: to see where this came from,[26] look at Eq. (6.91) and recall how we went from c_k to $\tilde{y}_k$ in Eq. (6.98).

A comment we will make for the fourth time in a row: Eq. (6.99) is simply transforming n numbers into n numbers. We keep repeating this fact in order to highlight that the direct and inverse DFT transforms are fundamentally discrete problems which refer to points on a grid. We will take this into consideration in what follows.

Fast Fourier Transform

As you will show in a problem, the case of real input data, y_j, is special, in that you only need to evaluate $m + 1$ Fourier parameters; this is so because $\tilde{y}_{n-k} = \tilde{y}_k^*$ holds for this case, so the remaining coefficients are complex conjugates of already-evaluated ones. (Incidentally, the $\tilde{y}_k$ might be complex, even when the y_j are all real; this is OK, because the inverse DFT of Eq. (6.99) makes everything real again.) You could consider this to be a way of reducing the computational runtime of a DFT implementation. Instead of following this approach, however, we will now turn to a much more effective way of speeding the DFT up, regardless of whether or not the input data, y_j, are real. As is usually the case in computing, algorithmic breakthroughs matter much more than "small efficiencies".

Before we start accelerating things, let us observe that our latest definition of the DFT,

[26] If you didn't follow our admonition to employ the orthogonality explicitly.

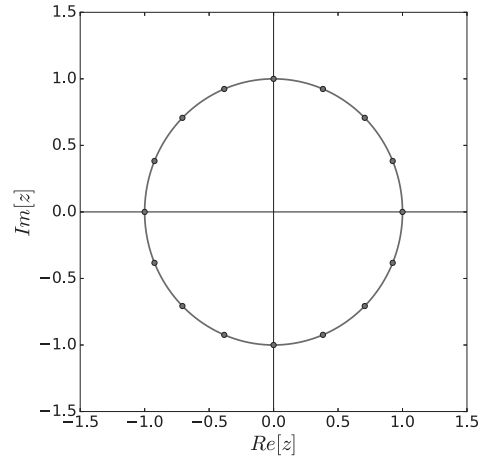


Fig. 6.10 Complex plane unit circle and 16th roots of unity, $e^{-2\pi ik/n}$ for $k = 0, 1, \ldots, 15$ and $n = 16$

Eq. (6.97), can be expressed in the form:

$$\tilde{y}_k = \sum_{j=0}^{n-1} (e^{-2\pi i/n})^{kj} y_j, \qquad k = 0, 1 \ldots, n-1 \tag{6.100}$$

But this, in its turn, can be written in matrix form:

$$\tilde{\mathbf{y}} = \mathbf{E}\mathbf{y} \tag{6.101}$$

where $\mathbf{E}$ is the $n \times n$ matrix made up of $(\mathbf{E})_{kj}$, which are powers of $e^{-2\pi i/n}$. In short, we now see that the problem of computing the DFT is essentially *matrix-vector multiplication*! As we discovered in section 4.3.1,[27] this is a problem for which the operation count is $O(n^2)$. The main idea behind the *fast Fourier transform* (FFT) is that the matrix-vector multiplication of Eq. (6.100) involves specific symmetries (despite not being sparse) and therefore does not have to cost $O(n^2)$. In other words, we will now see how to exploit the specific properties of the $\mathbf{E}$ matrix in order to make the DFT dramatically faster. For the sake of simplicity, we assume that n is a power of 2.

Let us first turn to an elementary problem involving complex numbers. We are looking for the roots of the equation $z^n = 1$; the solutions are known as the *n-th roots of unity*. Here's one of them:

$$e^{-2\pi i/n} = \cos\left(\frac{2\pi}{n}\right) - i \sin\left(\frac{2\pi}{n}\right) \tag{6.102}$$

Obviously, $(e^{-2\pi i/n})^n = 1$, meaning that $e^{-2\pi i/n}$ is, indeed, an n-th root of unity. It is not the only one, however. It should be easy to see that $(e^{-2\pi i/n})^k = e^{-2\pi ik/n}$, for $k = 0, 1, \ldots, n-1$, are all n-th roots of unity, since $(e^{-2\pi ik/n})^n = 1$ for all k's. For a given value of n, since k runs from 0 to $n-1$, there are n such distinct n-th roots of unity, $e^{-2\pi ik/n}$. They are illustrated in Fig. 6.10 for the case of $n = 16$. The crucial take-away here is that, for the case $n = 16$,

[27] Of course, here we are dealing with complex numbers, but that doesn't change the overall scaling.

there are 16 distinct roots *only*: even if you raise $e^{-2\pi ik/n}$ to another power, say j as in $(e^{-2\pi ik/n})^j$, you can only get back one of the 16 roots.[28]

Having grasped the meaning of the n-th roots of unity, we now return to our **E** matrix from Eq. (6.101). We realize that out of the n^2 matrix elements there are only n distinct ones (since many of them repeat); this **E** matrix is complex symmetric, but not Hermitian. To reiterate, the fast Fourier transform is essentially a clever way of exploiting the fact that our matrix-vector multiplication involves $(e^{-2\pi i/n})^{kj}$. It's easier to see this in a specific example, so let us write out Eq. (6.101) for the 4×4 case:

$$\begin{aligned}
\tilde{y}_0 &= y_0(e^{-2\pi i/4})^0 + y_1(e^{-2\pi i/4})^0 + y_2(e^{-2\pi i/4})^0 + y_3(e^{-2\pi i/4})^0 \\
\tilde{y}_1 &= y_0(e^{-2\pi i/4})^0 + y_1(e^{-2\pi i/4})^1 + y_2(e^{-2\pi i/4})^2 + y_3(e^{-2\pi i/4})^3 \\
\tilde{y}_2 &= y_0(e^{-2\pi i/4})^0 + y_1(e^{-2\pi i/4})^2 + y_2(e^{-2\pi i/4})^4 + y_3(e^{-2\pi i/4})^6 \\
\tilde{y}_3 &= y_0(e^{-2\pi i/4})^0 + y_1(e^{-2\pi i/4})^3 + y_2(e^{-2\pi i/4})^6 + y_3(e^{-2\pi i/4})^9
\end{aligned} \tag{6.103}$$

Were we to evaluate things at this stage, we would have to carry out $4^2 = 16$ multiplications. We notice that our exponents here go up to nine, even though we know that there are only four distinct roots of unity: $(e^{-2\pi i/4})^0$, $(e^{-2\pi i/4})^1$, $(e^{-2\pi i/4})^2$, and $(e^{-2\pi i/4})^3$. This means we can use elementary manipulations to re-express the 16 matrix elements of **E** in terms of only these four roots of unity. For example:

$$(e^{-2\pi i/4})^6 = (e^{-2\pi i/4})^4 (e^{-2\pi i/4})^2 = (e^{-2\pi i/4})^2 \tag{6.104}$$

Using this fact and regrouping, allows us to re-express our four equations in the form:

$$\begin{aligned}
\tilde{y}_0 &= [y_0(e^{-2\pi i/4})^0 + y_2(e^{-2\pi i/4})^0] + (e^{-2\pi i/4})^0[y_1(e^{-2\pi i/4})^0 + y_3(e^{-2\pi i/4})^0] \\
\tilde{y}_1 &= [y_0(e^{-2\pi i/4})^0 + y_2(e^{-2\pi i/4})^2] + (e^{-2\pi i/4})^1[y_1(e^{-2\pi i/4})^0 + y_3(e^{-2\pi i/4})^2] \\
\tilde{y}_2 &= [y_0(e^{-2\pi i/4})^0 + y_2(e^{-2\pi i/4})^0] + (e^{-2\pi i/4})^2[y_1(e^{-2\pi i/4})^0 + y_3(e^{-2\pi i/4})^0] \\
\tilde{y}_3 &= [y_0(e^{-2\pi i/4})^0 + y_2(e^{-2\pi i/4})^2] + (e^{-2\pi i/4})^3[y_1(e^{-2\pi i/4})^0 + y_3(e^{-2\pi i/4})^2]
\end{aligned} \tag{6.105}$$

In addition to factoring out and simplifying some of the exponents, observe that we first write the even y_j elements (y_0 and y_2) and then the odd ones (y_1 and y_3).

Introducing new notation, which we hope is self-explanatory, this takes the form:

$$\begin{aligned}
\tilde{y}_0 &= \tilde{y}_0^{\text{even}} + (e^{-2\pi i/4})^0 \tilde{y}_0^{\text{odd}} \\
\tilde{y}_1 &= \tilde{y}_1^{\text{even}} + (e^{-2\pi i/4})^1 \tilde{y}_1^{\text{odd}} \\
\tilde{y}_2 &= \tilde{y}_0^{\text{even}} + (e^{-2\pi i/4})^2 \tilde{y}_0^{\text{odd}} \\
\tilde{y}_3 &= \tilde{y}_1^{\text{even}} + (e^{-2\pi i/4})^3 \tilde{y}_1^{\text{odd}}
\end{aligned} \tag{6.106}$$

where we noticed that the y_j's and the powers of $e^{-2\pi i/4}$ inside the square brackets appear in only four distinct combinations, which we denoted by $\tilde{y}_0^{\text{even}}$, $\tilde{y}_0^{\text{odd}}$, $\tilde{y}_1^{\text{even}}$, and $\tilde{y}_1^{\text{odd}}$. To compute these, we need to carry out only $2 \times (4/2)^2 = 8$ multiplications. Make sure you understand what these entities are; for example:

$$\tilde{y}_1^{\text{even}} = y_0(e^{-2\pi i/4})^0 + y_2(e^{-2\pi i/4})^2 = y_0(e^{-2\pi i/2})^0 + y_2(e^{-2\pi i/2})^1 \tag{6.107}$$

[28] Once again, the similarity between Fig. 6.10 and Fig. 6.2 is hard to miss.

The $e^{-2\pi i/4}$ always appears raised either to the 0th or the 2nd power, so we can rewrite the exponential to have 2 in the denominator instead of 4. (This also applies to the $\tilde{y}^{\text{odd}}$'s.)

As per Eq. (6.106), once you have the two $\tilde{y}^{\text{even}}$'s and the two $\tilde{y}^{\text{odd}}$'s, you do four more multiplications in order to combine them together and produce the $\tilde{y}_k$. Even in this small problem, we have managed to reduce the total number of multiplications required.

We now take yet another step, rewriting Eq. (6.106) as follows:

$$\begin{aligned}
\tilde{y}_0 &= \tilde{y}_0^{\text{even}} + (e^{-2\pi i/4})^0 \tilde{y}_0^{\text{odd}} \\
\tilde{y}_1 &= \tilde{y}_1^{\text{even}} + (e^{-2\pi i/4})^1 \tilde{y}_1^{\text{odd}} \\
\tilde{y}_{2+0} &= \tilde{y}_0^{\text{even}} - (e^{-2\pi i/4})^0 \tilde{y}_0^{\text{odd}} \\
\tilde{y}_{2+1} &= \tilde{y}_1^{\text{even}} - (e^{-2\pi i/4})^1 \tilde{y}_1^{\text{odd}}
\end{aligned} \tag{6.108}$$

The first two relations are unchanged; the last two were rewritten to highlight the fact that all the relevant indices go from 0 to 1. On the right-hand sides of the last two equations we wrote $2 = 2 + 0$ and $3 = 2 + 1$ in the exponents and then made use of the fact that $e^{-\pi i} = -1$. In short, we have rewritten our 4×4 DFT problem from Eq. (6.103) in terms of two smaller (2×2) DFT problems, one for the even j indices and one for the odd j indices (recall Eq. (6.107)), which are combined with the appropriate prefactors as per Eq. (6.108).

Having worked out the 4×4 example in gory detail, we hope that you will now be able to understand without much effort the general case, which goes by the name of the *Danielson–Lanczos lemma*. Let's start from Eq. (6.97), splitting the sum over j into a sum over the even components, $j = 2l$, and the odd components, $j = 2l + 1$:

$$\begin{aligned}
\tilde{y}_k &= \sum_{j=0}^{n-1} y_j e^{-2\pi i k j/n} = \sum_{l=0}^{m-1} y_{2l} e^{-2\pi i k(2l)/n} + \sum_{l=0}^{m-1} y_{2l+1} e^{-2\pi i k(2l+1)/n} \\
&= \sum_{l=0}^{m-1} y_{2l} e^{-2\pi i k l/m} + e^{-2\pi i k/n} \sum_{l=0}^{m-1} y_{2l+1} e^{-2\pi i k l/m} = \tilde{y}_k^{\text{even}} + e^{-2\pi i k/n} \tilde{y}_k^{\text{odd}}
\end{aligned} \tag{6.109}$$

Note that in the second equality our sums switched to using l and therefore the maximum value became $m - 1$. In the exponents of the third equality we moved the 2 from the numerators to the denominators, thereby getting m; in the second term, we also factored out the odd term (which has an n in the denominator of the exponent). In the fourth equality we identified our new DFTs of length $m = n/2$, the ones for the even and odd components.

As we saw above, this is already a gain; however, let's take one more step, to make our general result match what we did for the 4×4 case. As you may recall, we managed to express the "second half" of our equations in terms of quantities that had already appeared in the first half. Right now, in Eq. (6.109) the k goes from 0 to $n - 1$, so we seem to be using more $\tilde{y}^{\text{even}}$'s and $\tilde{y}^{\text{odd}}$'s than we actually need. To remedy the situation, introduce a new wave number index q which runs from 0 to $m - 1$, i.e., it appears in the combination $k = m + q$. We have:

$$\tilde{y}_{m+q}^{\text{even}} = \sum_{l=0}^{m-1} y_{2l} e^{-2\pi i m l/m} e^{-2\pi i q l/m} = \sum_{l=0}^{m-1} y_{2l} e^{-2\pi i q l/m} = \tilde{y}_q^{\text{even}} \tag{6.110}$$

since $e^{-2\pi i l} = 1$. Now both the sum over l and the index q only cover half the range from 0

to $n-1$ (i.e., go from 0 to $m-1$). A fully analogous derivation can be carried out for the odd case, leading to:

$$\tilde{y}^{\text{odd}}_{m+q} = \tilde{y}^{\text{odd}}_{q} \tag{6.111}$$

There is only one other ingredient missing: the factored-out coefficients in Eq. (6.109) still employ k which goes up to $n-1$. As in Eq. (6.108), we can also halve those:

$$e^{-2\pi ik/n} = e^{-2\pi i(m+q)/n} = e^{-2\pi i/2} e^{-2\pi iq/n} = -e^{-2\pi iq/n} \tag{6.112}$$

Putting everything together, we have accomplished what we set out to do:

$$\begin{aligned} \tilde{y}_q &= \tilde{y}^{\text{even}}_q + e^{-2\pi iq/n} \tilde{y}^{\text{odd}}_q \\ \tilde{y}_{m+q} &= \tilde{y}^{\text{even}}_q - e^{-2\pi iq/n} \tilde{y}^{\text{odd}}_q \end{aligned} \tag{6.113}$$

where we left the first-half of the factored-out coefficients untouched. For both equations, we have $q = 0, 1, \ldots, m-1$. This closely matches our earlier result, Eq. (6.108).

As you may have suspected, the fast Fourier transform doesn't carry out such a halving process only once: it employs a *divide-and-conquer* approach, continually halving, until the problem becomes sufficiently small that it cannot be cut in half. At that point, the problem will be simple enough that the DFT can be trivially arrived at: you can see from Eq. (6.97) that when $n = 1$ we have $\tilde{y}_0 = y_0$. You may now realize why we said that our n has to be a power of 2: this was in order to enable us to keep halving until we get down to $n = 1$ without a problem. Each stage of this algorithm involves n multiplications and there are $\log_2 n$ such stages in total. In all, the fast Fourier transform operation count is $O(n \log_2 n)$, a dramatic improvement over the "naive" implementation of the DFT, which involved $O(n^2)$ operations. You may also recall, from chapter 1, that divide-and-conquer algorithms are most naturally implemented using recursion. This is fortunate, because it means that we do not have to produce separate notation for each stage of the algorithm: it is enough to show one halving step and state that we repeat the process recursively. We will return to these considerations soon.

As usual, things can be made even more efficient than that. You could think about where the different quantities are stored, how many of them can be re-used from one stage to the next, how you would go about reformulating the algorithm iteratively, and so on. However, the crucial point is that the FFT introduces a different *scaling*: the matrix multiplication of Eq. (6.101) is no longer quadratic! When n is large, this can have a dramatic effect on how long things take; a problem asks you to compare the runtime of DFT and FFT for the case of $n = 8192$. Viewed from another perspective, the FFT enables calculations that would have been impossible using only the old-fashioned DFT. This is one of the reasons why Fourier transforms are so prevalent in a very large number of applications. For example, in the study of differential equations, FFTs are routinely used to go to wave-number space and (trivially) take derivatives there (see section 8.5.2). Instead of following that route, however, we now recall that our goals in this chapter were more humble: we return to the problem of trigonometric interpolation.

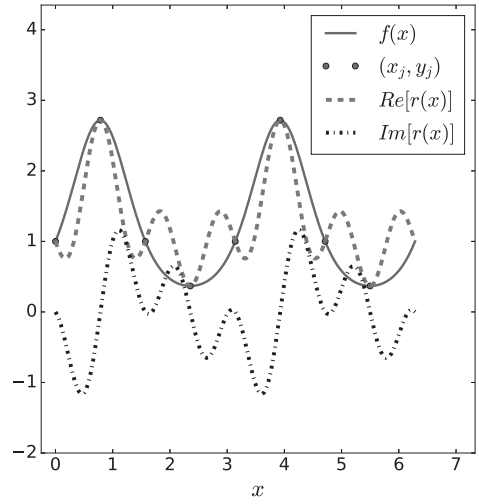


Fig. 6.11 Naive attempt to carry out shifted-DFT interpolation for the case of $n = 8$

Interpolation Using the FFT

Backing up for a second, we realize that the fast Fourier transform, Eq. (6.113), is just a fast way of producing the discrete Fourier transform, Eq. (6.97). In other words, both approaches simply evaluate the $\tilde{y}_k$'s for $k = 0, 1, \ldots, n-1$. A few subsections ago, once we calculated the a_k and b_k parameters we were ready to produce the interpolating polynomial $p(x)$ given in Eq. (6.82); similarly, in Eq. (6.88) we re-expressed $p(x)$ in terms of the c_k parameters. Wishing to produce the shifted-DFT analogue, once you've computed the $\tilde{y}_k$'s, you may be tempted to write down an interpolating polynomial of the form:

$$r(x) = \frac{1}{n} \sum_{k=0}^{n-1} \tilde{y}_k e^{ikx} \tag{6.114}$$

This bears a passing resemblance to our infinite Fourier series, Eq. (6.79), and looks pretty similar to our shifted inverse DFT, Eq. (6.99). As a matter of fact, it basically *is* Eq. (6.99) but this time evaluated not at $x_j = (2\pi j)/n$ but at any x. It is certainly plausible that $r(x)$ would do a good job interpolating, since we already know (by construction) that $r(x_j) = y_j$: evaluating Eq. (6.114) at the grid points turns $r(x)$ into the y values of our input data.

As you will discover when you solve the relevant problem, this polynomial doesn't actually do a very good job of interpolating between the (x_j, y_j) points; see Fig. 6.11 for the $n = 8$ case. Our $r(x)$ does go through the input data (x_j, y_j), as it should. However, comparing our new polynomial to our earlier trigonometric interpolation, see the right panel of Fig. 6.9, the general behavior is very different. Not only is the real part of $r(x)$ exhibiting wild fluctuations between the grid points, but $r(x)$ also comes with a sizable imaginary part, which also fluctuates. As it so happens, we already knew that the "interpolating" polynomial of Eq. (6.114) would give these different results, which is why we made sure to call it $r(x)$, i.e., not $p(x)$.

The reason $r(x)$ is so different from $p(x)$ has nothing to do with the $1/n$ that appears in Eq. (6.114). As you may recall, that was simply a standard choice in how to define the Fourier coefficients $\tilde{y}_k$ in comparison to our earlier c_k parameters. The $1/n$ should be there if you're using $\tilde{y}_k$ as input. The actual problems with using $r(x)$ as per Eq. (6.114) are two: (a) we assumed that a sum over non-negative k's, as per the shifted version of the DFT, would work, and (b) we didn't treat the largest/smallest frequency term, $\tilde{y}_m$, specially.

Both of these issues can be addressed in one go. Instead of producing an *ad hoc* guess, let us start with our interpolating polynomial $p(x)$ from Eq. (6.88) and update its definition to use our shifted Fourier coefficients, $\tilde{y}_k$. To refresh your memory, what we had there was:

$$p(x) = \sum_{k=-m+1}^{m-1} c_k e^{ikx} + c_{-m} \cos mx \tag{6.115}$$

We now recall that the correspondence between shifted and unshifted parameters was, as per Eq. (6.98):

$$\begin{aligned} \begin{pmatrix} \tilde{y}_0 & \tilde{y}_1 & \dots & \tilde{y}_{m-1} \end{pmatrix}^T &= n \begin{pmatrix} c_0 & c_1 & \dots & c_{m-1} \end{pmatrix}^T \\ \begin{pmatrix} \tilde{y}_m & \tilde{y}_{m+1} & \dots & \tilde{y}_{n-1} \end{pmatrix}^T &= n \begin{pmatrix} c_{-m} & c_{-m+1} & \dots & c_{-1} \end{pmatrix}^T \end{aligned} \tag{6.116}$$

This immediately leads us to the following formulation of our interpolating polynomial:

$$p(x) = \frac{1}{n} \sum_{k=0}^{m-1} \tilde{y}_k e^{ikx} + \frac{1}{n} \tilde{y}_m \cos mx + \frac{1}{n} \sum_{k=m+1}^{n-1} \tilde{y}_k e^{i(k-n)x} \tag{6.117}$$

where you will notice that we are using the same symbol, $p(x)$, since it *is* the same polynomial, this time expressed in terms of our shifted Fourier coefficients, $\tilde{y}_k$. The positive terms in the sum remain unchanged, the negative ones are handled separately, as is true of the midpoint (which corresponds to the largest/smallest wave number). The only subtle point here is the exponent in the second sum: to match the exponents for the negative terms of the sum in Eq. (6.115), we need to shift; make sure to check that $k - n$ goes from $-m + 1$ to -1, when k goes from $m + 1$ to $n - 1$. Finally, we are also including a $1/n$ term to ensure the two definitions match.

Implementation

We are now ready to implement trigonometric interpolation using the FFT. If you've been following along, you will realize that we could have done something analogous at several points along the way: we could have implemented the unshifted DFT, Eq. (6.94), to find the c_k's and then used that to produce $p(x)$ as per Eq. (6.88). We could have also implemented the shifted DFT, Eq. (6.97), to find the $\tilde{y}_k$'s and then computed $p(x)$ from Eq. (6.117). Both of these are easier programming tasks, which you are asked to carry out in the problem set. Instead, we will now capitalize on all our progress so far and implement the FFT, Eq. (6.113), to find the $\tilde{y}_k$'s and then plug those in to Eq. (6.117) to compute $p(x)$. Since we

Code 6.4 fft.py

```
from triginterp import f, generatedata
from math import pi
import numpy as np

def fft(ys):
     n = ys.size
     m = n//2
     if n==1:
          ytils = ys
     else:
          evens = fft(ys[::2])
          odds = fft(ys[1::2])
          coeffs = np.exp(-2*pi*np.arange(m)*1j/n)
          first = evens + coeffs*odds
          second = evens - coeffs*odds
          ytils = np.concatenate((first, second))
     return ytils

def fftinterp(ytils,x):
     n = ytils.size
     m = n//2
     val = ytils[:m]@np.exp(np.arange(m)*x*1j)
     val += ytils[m]*np.cos(m*x)
     val += ytils[m+1:]@np.exp(np.arange(-m+1,0)*x*1j)
     return val/n

if __name__ == '__main__':
     n = 8
     dataxs, datays = generatedata(n, f)
     ytils = fft(datays)
     x = 0.3; pofx = fftinterp(ytils, x)
     print(x,pofx.real)
```

are faced with two separate tasks, computing the $\tilde{y}_k$'s and $p(x)$, we define two functions in Code 6.4. Our first function implements the fast Fourier transform. As discussed in chapter 1, and around Eq. (6.113), it is natural to code up divide-and-conquer algorithms using recursion. We start from our *base case*, $n = 1$, for which the answer can be directly

given: we know from Eq. (6.97) that in this case we have $\tilde{y}_0 = y_0$. For any other value of n (always a power of 2), we cut the problem up into two halves using `numpy`'s slicing: `ys[::2]` and `ys[1::2]` select the even and odd components, respectively. Then, `fft()` calls itself to solve that simpler problem. Once an answer for `evens` and `odds` is known, our program simply multiplies with $+e^{-2\pi iq/n}$ or $-e^{-2\pi iq/n}$, exactly as in Eq. (6.113). It's interesting to note that we are not employing any loops, so the q index doesn't explicitly appear. We combine our two halves together by using `numpy.concatenate()` to make a larger array. In all, even though we are implementing a state-of-the-art algorithm, this function is very short and reasonably straightforward to understand. You may wish to print out its output or intermediate values, in order to be certain that everything is clear to you.

Specifically, you should make sure to understand that, even though Eq. (6.113) is written in terms of q and $m+q$, the end result is that we've managed to compute the $\tilde{y}_k$ coefficients, with k going from 0 to $n-1$. This may be even easier to grasp when you solve the problem that asks you to evaluate the $\tilde{y}_k$'s using the (slow) DFT approach, Eq. (6.97). Having computed the $\tilde{y}_k$'s, our second function uses them to implement Eq. (6.117). Nothing exciting is going on here: once again, we are employing `numpy` functionality, i.e., the dot product of one-dimensional arrays using `@`, to avoid having explicit loops and indices. Our three lines of code directly correspond to the three terms on the right-hand side of Eq. (6.117).

The main program first creates some data points using the functions we defined in `triginterp.py`. It then calls our two new functions to compute the $\tilde{y}_k$'s and then the interpolating polynomial $p(x)$. Once again, we pick a value of x at random, just to show that we can evaluate $p(x)$ at any point we choose. We end by printing out the real part of the value of $p(x)$; crucially, the imaginary part that we are dropping is *always* zero. Our $\tilde{y}_k$ and the exponentials in Eq. (6.117) are complex, but they combine to produce a real interpolating polynomial at any value of x. This is so by construction, since we've been trying to match our trigonometric-interpolation polynomial every step of the way. As a result, we can use Code 6.4 to reproduce the right panel of Fig. 6.9 exactly.

6.5 Least-Squares Fitting

As we saw in our introduction in section 6.1.2, *least-squares fitting* is different from what we've spent the last several pages doing: we will no longer trust our input table (x_j, y_j) of N data pairs blindly. This time around, each data point will have an associated measurement error, σ_j; you may wish to think of the y-values of your data as $y_j \pm \sigma_j$. In other words, we will *not* demand that our approximating function, $p(x)$, go through the data points; instead, we will try to come up with a $p(x)$ that "roughly" captures the behavior of the data. Of course, there are infinitely many choices we could make; in practice, one is guided by other knowledge of the system, say from the physical theory that describes that observable. If no extra knowledge is available, one typically picks a $p(x)$ that is a polynomial in x, but this time a low-degree polynomial should be enough, since the higher the degree the larger the number of undetermined parameters: since we don't fully trust the input data, we don't want to capture the "scatter", but only the underlying trend. An example of this was shown

in Fig. 6.1, where we had assumed that the data points could not be fully trusted, and taken the overall trend to be roughly linear.

6.5.1 Chi Squared

Let's study this problem a bit more systematically. Recall that we are still expanding our $p(x)$ in terms of n parameters c_k, as per Eq. (6.7):

$$p(x) = \sum_{k=0}^{n-1} c_k \phi_k(x) \tag{6.118}$$

where the basis functions $\phi_k(x)$ may be polynomials (or not). This is a *linear* problem, in the sense that $p(x)$ is a linear combination of the c_k's. In the context of least-squares fitting, our approximating function $p(x)$ is sometimes known as a *theory* or a *model*.

As we mentioned around Eq. (6.10), we have N data points and n undetermined parameters (where $N > n$), so this could be thought of as an *overdetermined system*. As we saw there, since $\mathbf{\Phi c} = \mathbf{y}$ cannot be solved exactly, what we'll do is the closest thing available, i.e., we'll try to minimize the norm of the residual vector, as per Eq. (6.11):

$$\min \|\mathbf{\Phi c} - \mathbf{y}\| \tag{6.119}$$

Of course, this task seems to only depend on the basis functions and the input data: where did the input uncertainties σ_j go? Qualitatively, what this is asking is: suppose some of your input data have large uncertainties, and perhaps also appear to show different behavior than the rest. Obviously, the points with larger measurement error should somehow be *weighted* differently, i.e., they shouldn't impact our final determination as much as points that were measured much more accurately.

Since we wish to take the measurement errors into account, we choose to minimize:

$$\chi^2 = \sum_{j=0}^{N-1} \left(\frac{y_j - p(x_j)}{\sigma_j} \right)^2 \tag{6.120}$$

which is known as the *chi squared*. It goes without saying (but we'll say it anyway) that (x_j, y_j) are variables (independent and dependent, respectively) whereas the c_k's are parameters; in other words, we'll never *measure* the c_k's directly. Note that this sum extends over all the N data points; the approximating function Eq. (6.118) is (implicitly) dependent on the values of the parameters c_k. When a given measurement error is large, the contribution of that term in the sum is small, as it should be. Presumably you can now see where the name *least-squares fitting* comes from: as shown by Eq. (6.120), minimizing χ^2 minimizes the distance between the theory and the data ($p(x_j)$ and y_j, respectively), weighted by the size of the error bar in the data.

Since we will choose the c_k parameters that *minimize* χ^2, we are free to take the derivative of Eq. (6.120) with respect to a given parameter c_k and set the result to zero:

$$\frac{\partial \chi^2}{\partial c_k} = -2 \sum_{j=0}^{N-1} \left(\frac{y_j - p(x_j)}{\sigma_j^2} \right) \frac{\partial p(x_j)}{\partial c_k} = 0 \tag{6.121}$$

where $k = 0, 1, \ldots, n-1$. Observe that in $\partial p(x)/\partial c_k$ we are taking the derivative with respect to c_k, since x_j and y_j are considered externally given (and "frozen"). Assuming that the approximating function $p(x)$ is linear in the c_k's, as per Eq. (6.118), means that these derivatives can be trivially taken, as we'll see below: this leads to n linear equations in n unknowns, namely the c_k's.

If we now make the further assumption that the σ_j's are normally distributed, then our prescription above provides *maximum likelihood parameter estimation* for the c_k's. While the subject of *statistical inference* is quite interesting, our goals here are more limited: we try to find the c_k's that mimize the χ^2, without worrying too much about questions regarding goodness-of-fit.[29] Even so, we would like to have some guidance on the question of how to compare different theories. To reiterate, minimizing the χ^2 of Eq. (6.120) tells you how to find the best c_k's you can, *for a given theory*. But what if your theory is wrong? For example, imagine that you should have picked $p(x) = c_0 + c_1 x + c_2 x^2$ but instead you picked $p(x) = c_0 + c_1 x$. The χ^2-minimization process still tries to do its best, but the results may end up being poor, as you could see simply by inspection (i.e., by comparing $p(x)$ to the input data: it is rare that a parabola can be approximated by a straight line).

Here, we give some practical advice that will help you decide which theory to pick, always under the assumption that the σ_j's are normally distributed. If the (minimum) χ^2 is very large, you can see from its definition in Eq. (6.120) that you have not been able to produce a $p(x)$ that goes "near" the data points; this could simply be due to the fact that your model is wrong. Another explanation could be that whoever produced the σ_j's underestimated them, i.e., they are actually much larger, in which case your model would do a better job since the χ^2 would be smaller. On the other hand, if $\chi^2 \approx 0$, then from the same definition, Eq. (6.120), you see that your theory curve essentially goes through the data points. This is not always a good outcome: you may be using too many parameters, thereby "overfitting", i.e., trying to capture the random scatter in the data. Another explanation could be, as before, related to whoever produced the σ_j's: perhaps these were overestimated, i.e., in reality the σ_j's should be much smaller. As always, the question arises how to distinguish between "large" and "small" chi-squared values. As a rule of thumb, typically $\chi^2 \approx N - n$ implies a reasonably good fit.[30] The difference between the number of data points and the number of parameters, $N - n$, is known as the number of *degrees of freedom*. Thus, the rule of thumb can be re-stated as saying that for a fit that is not too good nor too bad you should expect the *chi-squared per degree of freedom to be roughly one*.

It turns out that the quantity to be minimized in Eq. (6.120) is equivalent to what we were faced with in Eq. (6.119), only this time we're also appropriately dividing with the

[29] Which means that we also don't go into important alternative scenarios like Poisson-distributed data.

[30] If you were not given the σ_j's as input, you could assume that they are all equal, $\sigma_j = \sigma$, and also that you did a good job fitting, in which case $\chi^2 \approx N - n$. Then, you could find σ^2 from Eq. (6.120) but, of course, this does not help you check how well you did in your fit, since you started by assuming you did well.

input uncertainties. We will return to the matrix notation below but, for now, we will help you build some intuition on least-squares fitting, by showing Eq. (6.121) explicitly applied to a simple case.

6.5.2 Straight-Line Fit

You have probably encountered the task of fitting data to a straight line in an introductory lab course. Here, we will essentially repeat what you saw there, employing the notation that we introduced above; in the following section, we go over a more general form, which will map the least-squares problem to a linear-algebra one.

Our task is to fit N data points (x_j, y_j) to a model which is a straight line:

$$p(x) = c_0 + c_1 x \tag{6.122}$$

We need to determine the 2 parameters c_0 and c_1. For this theory, the chi squared definition of Eq. (6.120) takes the form:

$$\chi^2 = \sum_{j=0}^{N-1} \left(\frac{y_j - c_0 - c_1 x_j}{\sigma_j} \right)^2 \tag{6.123}$$

where we are explicitly showing the dependence on the c_k parameters.

Parameter Estimates and Variances

The vanishing-derivative condition of Eq. (6.121) for our theory, Eq. (6.122), gives:

$$\frac{\partial \chi^2}{\partial c_0} = -2 \sum_{j=0}^{N-1} \left(\frac{y_j - c_0 - c_1 x_j}{\sigma_j^2} \right) = 0, \qquad \frac{\partial \chi^2}{\partial c_1} = -2 \sum_{j=0}^{N-1} \left(\frac{y_j - c_0 - c_1 x_j}{\sigma_j^2} \right) x_j = 0 \tag{6.124}$$

where the derivatives of $p(x_j)$ with respect to each c_k were easy to produce. Each numerator can be split into three terms; to do so, let us define the following six helper variables:

$$\begin{aligned} S &\equiv \sum_{j=0}^{N-1} \frac{1}{\sigma_j^2}, & S_x &\equiv \sum_{j=0}^{N-1} \frac{x_j}{\sigma_j^2}, & S_y &\equiv \sum_{j=0}^{N-1} \frac{y_j}{\sigma_j^2}, \\ S_{xx} &\equiv \sum_{j=0}^{N-1} \frac{x_j^2}{\sigma_j^2}, & S_{xy} &\equiv \sum_{j=0}^{N-1} \frac{x_j y_j}{\sigma_j^2}, & \Delta &\equiv S S_{xx} - S_x^2 \end{aligned} \tag{6.125}$$

Note that all the right-hand sides here are expressed in terms of known quantities, which means that they can be computed from the data. We can now re-express our vanishing-derivative results from Eq. (6.124) in terms of these helper variables:

$$\begin{aligned} S c_0 + S_x c_1 &= S_y \\ S_x c_0 + S_{xx} c_1 &= S_{xy} \end{aligned} \tag{6.126}$$

This is a simple 2×2 linear system of equations:

$$\begin{pmatrix} S & S_x \\ S_x & S_{xx} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} S_y \\ S_{xy} \end{pmatrix} \tag{6.127}$$

If you're wondering why we went through the trouble of defining the Δ, the answer is that it helps us compactly write down the solution to our 2×2 system:

$$c_0 = \frac{S_{xx}S_y - S_x S_{xy}}{\Delta}, \qquad c_1 = \frac{S S_{xy} - S_x S_y}{\Delta} \tag{6.128}$$

Since the helper variables of Eq. (6.125) were computed in terms of the data, everything here is known: we have managed to find the c_0 and c_1 values that minimize the χ^2.

At this point, we could immediately proceed to a Python implementation of Eq. (6.128) for a given set of data. Instead, let us take a minute to think about the values of c_0 and c_1 we just determined. We recall that these are *estimates* of the parameter values: since our input data suffered from measurement errors, σ_j, it is obvious that a corresponding uncertainty has been introduced in the determination of the c_k parameters. In short, we now need to make a quick detour through the subject of *propagation of error* which, as you may recall, is a topic we spent quite a bit of time discussing all the way back in section 2.2.2; however, back then we were referrring to "maximal error" propagation, whereas we are now interested in employing "standard error" propagation. In short, this time around we will add the errors in quadrature, in contradistinction to what we did, say, in Eq. (2.41). Assuming the data are independent, propagation of error for any function g takes the form:

$$\sigma_g^2 = \sum_{j=0}^{N-1} \left(\frac{\partial g}{\partial y_j} \right)^2 \sigma_j^2 \tag{6.129}$$

where we have assumed the data are independent and in our specific case g will be either c_0 or c_1. We can evaluate the needed derivatives from our answer in Eq. (6.128):

$$\frac{\partial c_0}{\partial y_j} = \frac{S_{xx} - S_x x_j}{\sigma_j^2 \Delta}, \qquad \frac{\partial c_1}{\partial y_j} = \frac{S x_j - S_x}{\sigma_j^2 \Delta} \tag{6.130}$$

It may help you to use different dummy summation variables in the definitions of our helpers, Eq. (6.125), to see where these results came from. The first equation helps us determine $\sigma_{c_0}^2$ and the second one $\sigma_{c_1}^2$. Plugging these two equations into Eq. (6.129), expanding the parentheses, and collecting terms leads to:

$$\sigma_{c_0}^2 = \frac{S_{xx}}{\Delta}, \qquad \sigma_{c_1}^2 = \frac{S}{\Delta} \tag{6.131}$$

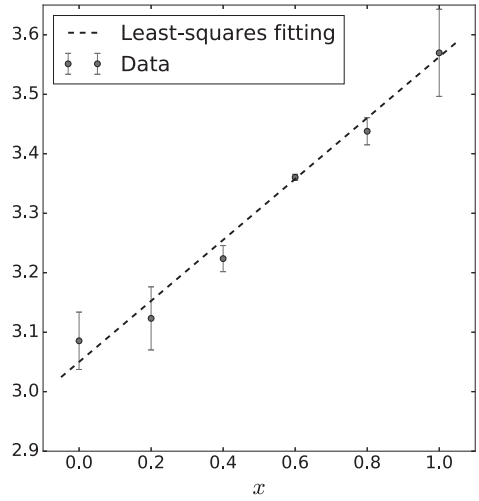


Fig. 6.12 Least-squares fitting for a straight-line theory

These are the *variances* in the estimates for our two parameters. If you want the *standard deviations*, you'll have to take the square root(s). As you will see in a problem, one can also compute the *covariance* of c_0 and c_1, but what we have here is enough for our purposes.

Implementation

To see our least-squares straight-line fitting algorithm at work, we'll need a set of input data. We decide to re-use the data shown in Fig. 6.1: the measurement errors, σ_j, were suppressed there (since interpolation doesn't need them), but can be taken into account now. Code 6.5 first computes the helper variables of Eq. (6.125). We then define another function to evaluate the c_k's from Eq. (6.128) and the σ_{c_k}'s from Eq. (6.131). We store these in two `numpy` arrays: this is overkill for our 2×2 problem, but it sets the stage for what is to follow.[31] Having determined the c_k's, we then define another function to compute the (minimum) χ^2 from Eq. (6.123).

The main program sets up the data corresponding to Fig. 6.1 and then proceeds to call the last two functions, first computing the c_k's and then using them to compute the χ^2. Printed out are the parameters and their standard deviations, as well as the χ^2 per degree of freedom, i.e., $\chi^2/(N-n)$. The output of running this program is:

```
[ 3.04593186 0.5189044 ]
[ 0.02927752 0.04896135]
1.09916819554
```

We see that the χ^2 per degree of freedom is pretty close to 1, signifying a decent fit. This

[31] Note that we could have opted in favor of *not* solving the 2×2 system of Eq. (6.127) by hand: instead, we could have simply set it up and then used Gaussian elimination to solve it programmatically.

linefit.py Code 6.5

```
import numpy as np

def helpers(dataxs,datays,datasigs):
     S = np.sum(1/datasigs**2)
     Sx = np.sum(dataxs/datasigs**2)
     Sy = np.sum(datays/datasigs**2)
     Sxx = np.sum(dataxs**2/datasigs**2)
     Sxy = np.sum(dataxs*datays/datasigs**2)
     Del = S*Sxx - Sx**2
     return S, Sx, Sy, Sxx, Sxy, Del

def computecs(dataxs,datays,datasigs):
     S,Sx,Sy,Sxx,Sxy,Del = helpers(dataxs,datays,datasigs)
     cs = np.zeros(2); dcs = np.zeros(2)
     cs[0] = (Sxx*Sy - Sx*Sxy)/Del
     cs[1] = (S*Sxy - Sx*Sy)/Del
     dcs[0] = np.sqrt(Sxx/Del)
     dcs[1] = np.sqrt(S/Del)
     return cs, dcs

def computechisq(dataxs,datays,datasigs,cs):
     chisq = np.sum((datays-cs[0]-cs[1]*dataxs)**2/datasigs**2)
     return chisq

dataxs = np.linspace(0,1,6)
datays = np.array([3.085, 3.123, 3.224, 3.360, 3.438, 3.569])
datasigs = np.array([0.048, 0.053, 0.02, 0.005, 0.023, 0.07])
cs, dcs = computecs(dataxs, datays, datasigs)
print(cs); print(dcs)
chisq = computechisq(dataxs, datays, datasigs, cs)
print(chisq/(dataxs.size - cs.size))
```

is also reflected in the error bars we have placed on the parameters. Speaking of which, we observe that the error in the offset parameter, c_0, is much smaller than that in the slope, c_1.

In Fig. 6.12 we are showing the data along with the measurement errors, σ_j. We're also plotting the straight-line model, Eq. (6.122), that results from our best-fit parameters c_0 and c_1.[32] Even if you take the error bars into consideration, the straight-line model does

[32] We could have also used the σ_{c_k}'s and thereby produced a *band* instead of a straight line.

not need to go through all the points. We are minimizing the χ^2 from Eq. (6.123), trying to take into account all the data points and associated errors; our procedure does the best job it can, always under the assumption that our model is a straight line.

6.5.3 General Linear Fit: Normal Equations

The (standard) material that we developed in the previous section is good as far as it goes. However, it's easy to see that solving things "by hand" doesn't take you very far. For example, imagine you wanted to check whether you picked the right model. One way to do so would be to try a quadratic theory:

$$p(x) = c_0 + c_1 x + c_2 x^2 \tag{6.132}$$

And repeat the entire least-squares fitting procedure.[33] This involves three unknown parameters; in this case Eq. (6.127) generalizes to:

$$\begin{pmatrix} S & S_x & S_{xx} \\ S_x & S_{xx} & S_{xxx} \\ S_{xx} & S_{xxx} & S_{xxxx} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} S_y \\ S_{xy} \\ S_{xxy} \end{pmatrix} \tag{6.133}$$

where you should be able to guess how the new terms like S_{xxxx} are defined. A problem asks you to derive this, starting from the zero-derivative conditions in Eq. (6.121).

A moment's thought will convince you that the coefficient matrix here is an extension of the Vandermonde matrix we encountered in Eq. (6.17). As you may recall from that discussion, such matrices can be very ill-conditioned, especially for large dimensions. Since Lagrange interpolation for $n = 100$ or even $n = 1000$ is routinely used in practice, this is a real issue. However, when carrying out *least-squares fitting*, you would generally not like to use a polynomial of very high degree: as mentioned in section 6.5.1, this mostly leads to overfitting, namely to capturing the random scatter in the data, which is an unwelcome outcome. In other words, you will typically not get into too much trouble if you attack the 3×3 problem of Eq. (6.132) via the naive approach of Eq. (6.133).[34] Of course, what happens if you then decide to try out the 4×4 or the 5×5 case? It's easy to see that you will have to give up on solving the linear system analytically, opting for Gaussian elimination or another method, instead.

So far in this subsection we have been considering the possibility of employing a model that is a polynomial, raising the question of what order we should go up to. A related question arises when your basis functions are not monomials/polynomials. For example, your model could be:

$$p(x) = c_0 + c_1 e^{-x^2} \tag{6.134}$$

Note that this is still linear (in the c_k's), as per Eq. (6.118); it's only one of the basis

[33] As you will see in a problem, in this case you find that the coefficient of the quadratic term is very small, thereby increasing your confidence in your decision to carry out a straight-line fit for this set of data.

[34] Even so, one of the problems introduces an alternative approach, involving a generalization of the QR decomposition we encountered in chapter 4 to rectangular matrices; this leads to a more robust prescription, which is routinely used in libraries.

functions that is not linear (in x). As you will discover in one of the problems, trying to carry out a least-squares fit for this model will lead you to introduce helper variables like:

$$S_x \equiv \sum_{j=0}^{N-1} \frac{e^{-x_j^2}}{\sigma_j^2} \tag{6.135}$$

in direct generalization of what we saw in the straight-line case. Obviously, if you had three (or, say, seven) parameters, this way of doing things "by hand" would no longer be very practical.

Normal Equations

Instead of studying models on a case-by-case basis, we will now tackle the general (linear) problem of Eq. (6.118):

$$p(x) = \sum_{k=0}^{n-1} c_k \phi_k(x) \tag{6.136}$$

It should be easy to see that polynomial fitting is merely a special case of this model, that in which the $\phi_k(x)$'s are monomials. Thus, we will develop and implement the general machinery for any set of basis functions; then, in one of the problems you will specialize our code to the case of polynomial fitting.

For this model, the χ^2 of Eq. (6.120) takes the form:

$$\chi^2 = \sum_{j=0}^{N-1} \frac{1}{\sigma_j^2} \left(y_j - \sum_{i=0}^{n-1} c_i \phi_i(x_j) \right)^2 \tag{6.137}$$

Similarly, the zero-derivative conditions of Eq. (6.121) now become:

$$\frac{\partial \chi^2}{\partial c_k} = -2 \sum_{j=0}^{N-1} \frac{1}{\sigma_j^2} \left(y_j - \sum_{i=0}^{n-1} c_i \phi_i(x_j) \right) \phi_k(x_j) = 0 \tag{6.138}$$

where $k = 0, 1, \ldots, n-1$, as usual. If we now drop the -2 coefficient, re-arrange terms, and interchange the order of summation, we find:

$$\sum_{i=0}^{n-1} \sum_{j=0}^{N-1} \frac{\phi_k(x_j)}{\sigma_j} \frac{\phi_i(x_j)}{\sigma_j} c_i = \sum_{j=0}^{N-1} \frac{\phi_k(x_j)}{\sigma_j} \frac{y_j}{\sigma_j} \tag{6.139}$$

Keep in mind that typically N is large and n is small. On both sides, we are summing over the j index, which corresponds to the (large) number of data points, N. The way we have chosen to write Eq. (6.139) clearly motivates the introduction of two new entities:

$$A_{jk} = \frac{\phi_k(x_j)}{\sigma_j}, \qquad b_j = \frac{y_j}{\sigma_j} \tag{6.140}$$

You can see that $\mathbf{A} = \{A_{jk}\}$ is a rectangular matrix made up of $N \times n$ elements; the row index goes from 0 to $N - 1$ and the column index from 0 to $n - 1$. This is usually called

the *design matrix* of the problem. Similarly, $\mathbf{b}$ is an $N \times 1$ column vector, made up of the data scaled by the measurement errors. Crucially, all the elements of $\mathbf{A}$ and $\mathbf{b}$ can be immediately computed from the input data.

In terms of our two new entities, Eq. (6.139) becomes:

$$\sum_{i=0}^{n-1} \sum_{j=0}^{N-1} A_{jk} A_{ji} c_i = \sum_{j=0}^{N-1} A_{jk} b_j \tag{6.141}$$

We now recognize that the sums over j can be replaced by matrix–matrix multiplication (on the left-hand side) and matrix-vector multiplication (on the right-hand side):

$$\sum_{i=0}^{n-1} (\mathbf{A}^T \mathbf{A})_{ki} c_i = (\mathbf{A}^T \mathbf{b})_k \tag{6.142}$$

where we appropriately introduced the transpose of $\mathbf{A}$ whenever we needed two indices to appear in different order. We now realize that even the summation over i on the left-hand side can be expressed in terms of a matrix-vector multiplication, leading us to write:

$$(\mathbf{A}^T \mathbf{A}\mathbf{c})_k = (\mathbf{A}^T \mathbf{b})_k \tag{6.143}$$

But since this holds for any component k, we have thereby shown:

$$\mathbf{A}^T \mathbf{A}\mathbf{c} = \mathbf{A}^T \mathbf{b} \tag{6.144}$$

Given that $\mathbf{A}$ and $\mathbf{b}$ are already known (computed in terms of the input data), this equation is a simple linear system of equations: we can solve it using, say, Gaussian elimination to find $\mathbf{c}$. Make sure you understand the dimensions of each entity: $\mathbf{A}$ is $N \times n$, which means $\mathbf{A}^T$ is $n \times N$ and therefore $\mathbf{A}^T \mathbf{A}$ is $n \times n$. This is appropriate since $\mathbf{c}$ is an $n \times 1$ vector containing all the c_k's. Similarly, $\mathbf{A}^T$ is $n \times N$ and $\mathbf{b}$ is an $N \times 1$, so $\mathbf{A}^T \mathbf{b}$ is $n \times 1$. In other words, we have managed to transform our problem into square form, with a coefficient matrix of dimension $n \times n$: this is in general a much easier problem than dealing with $\mathbf{A}$ directly (which has dimensions $N \times n$), since typically n will be much smaller than N.

Interpreting the Normal Equations

Let us now build some intuition on the $\mathbf{A}^T \mathbf{A}$ matrix. Our main goal is to show that the $\mathbf{c}$ that we get from Eq. (6.144) is not only a critical point, but a *minimum*. You might want to brush up on the multidimensional minimization material in section 5.5.2 at this point.

First, it's easy to see that $\mathbf{A}^T \mathbf{A}$ is symmetric:

$$(\mathbf{A}^T \mathbf{A})^T = \mathbf{A}^T (\mathbf{A}^T)^T = \mathbf{A}^T \mathbf{A} \tag{6.145}$$

Since $\mathbf{A}^T \mathbf{A}$ is equal to its transpose, it is symmetric. Second, $\mathbf{A}^T \mathbf{A}$ is positive semidefinite, because for any non-trivial vector $\mathbf{x}$ we have:

$$\mathbf{x}^T \mathbf{A}^T \mathbf{A}\mathbf{x} = (\mathbf{A}\mathbf{x})^T \mathbf{A}\mathbf{x} = \|\mathbf{A}\mathbf{x}\|^2 \geq 0 \tag{6.146}$$

where, as usual, we're implicitly using the Euclidean norm. We can do even better: write $\mathbf{A}$ in terms of its columns:

$$\mathbf{A} = \begin{pmatrix} \mathbf{a}_0 & \mathbf{a}_1 & \dots & \mathbf{a}_{n-1} \end{pmatrix} \tag{6.147}$$

If we assume that the columns of $\mathbf{A}$ are linearly independent, we can re-express $\mathbf{Ax}$, whose norm we were taking in Eq. (6.146), as:

$$\mathbf{Ax} = \sum_{i=0}^{n-1} \mathbf{a}_i x_i \neq 0 \tag{6.148}$$

where the last step follows from the definition of being linearly independent. That means that we have shown $\|\mathbf{Ax}\|^2 \neq 0$, which implies $\|\mathbf{Ax}\|^2 > 0$, i.e., $\mathbf{A}^T\mathbf{A}$ is positive definite if the columns of $\mathbf{A}$ are linearly independent.

We now realize that our new $\mathbf{A}$ and $\mathbf{b}$ entities from Eq. (6.140) can help us rewrite the χ^2 from Eq. (6.137) in matrix form:

$$\chi^2(\mathbf{c}) = (\mathbf{b} - \mathbf{Ac})^T(\mathbf{b} - \mathbf{Ac}) \tag{6.149}$$

where we also took the opportunity to explicitly note that χ^2 depends on the values of the c_k parameters. To see that $\chi^2(\mathbf{c})$ involves a quadratic form in $\mathbf{c}$, expand things out:

$$\chi^2(\mathbf{c}) = \mathbf{b}^T\mathbf{b} - \mathbf{b}^T\mathbf{Ac} - \mathbf{c}^T\mathbf{A}^T\mathbf{b} + \mathbf{c}^T\mathbf{A}^T\mathbf{Ac} \tag{6.150}$$

In a problem, you will show that the Hessian of a quadratic form $\mathbf{x}^T\mathbf{Bx}$ is $\mathbf{H} = 2\mathbf{B}$ when $\mathbf{B}$ is symmetric. Recall that the Hessian for our case would be defined by analogy to Eq. (5.98):

$$H_{ij} = \frac{\partial\chi^2(\mathbf{c})}{\partial c_i \partial c_j} \tag{6.151}$$

From Eq. (6.150) you can see that only the quadratic form $\mathbf{c}^T\mathbf{A}^T\mathbf{Ac}$ contributes to the second derivative of $\chi^2(\mathbf{c})$; thus, $\mathbf{H} = 2\mathbf{A}^T\mathbf{A}$ for our case. The reason we care about the Hessian is that it makes an appearance in the multidimensional Taylor expansion (see Eq. (5.93)):

$$\chi^2(\mathbf{c} + \mathbf{q}) = \chi^2(\mathbf{c}) + \left(\nabla\chi^2(\mathbf{c})\right)^T \mathbf{q} + \frac{1}{2}\mathbf{q}^T\mathbf{Hq} \tag{6.152}$$

where now there is no term $O(\|\mathbf{q}\|^3)$ since the third derivative of a quadratic form vanishes. Recall that our main result in Eq. (6.144) gives us the $\mathbf{c}$ value that corresponds to the zero-derivative conditions, Eq. (6.138). In other words, it is the analogue of Eq. (5.97):

$$\nabla\chi^2(\mathbf{c}^*) = \mathbf{0} \tag{6.153}$$

This $\mathbf{c}^*$ is the set of c_k's that satisfies Eq. (6.144).[35] But this means that the vanishing gradient leads to:

$$\chi^2(\mathbf{c}^* + \mathbf{q}) = \chi^2(\mathbf{c}^*) + \frac{1}{2}\mathbf{q}^T\mathbf{Hq} > \chi^2(\mathbf{c}^*) \tag{6.154}$$

which is essentially a repeat of our argument in Eq. (5.100). The last step, showing that $\mathbf{c}^*$

[35] If you're careful, you could simply *derive* Eq. (6.144) by taking the gradient of Eq. (6.150).

is, indeed, a minimum as desired, follows from the fact that our $\mathbf{H}$ is positive definite. This is all consistent with what we said in section 5.5.2: a necessary condition for $\mathbf{c}^*$ being a local minimum is that it be a critical point and a sufficient condition for the critical point $\mathbf{c}^*$ being a local minimum is that its Hessian matrix be positive definite.

Before concluding, let us return to the cases where the normal-equations approach can get in trouble (and therefore another algorithm should be preferred). The main takeaway from the last few pages has been that Eq. (6.144) has transformed the problem such that we are no longer faced with the rectangular $\mathbf{A}$ matrix but with the square $\mathbf{A}^T\mathbf{A}$ matrix. As you will explicitly show in a problem, the condition numbers of the two matrices are straightforwardly related:

$$\kappa(\mathbf{A}^T\mathbf{A}) \approx \kappa(\mathbf{A})^2 \tag{6.155}$$

Thus, if $\mathbf{A}$ is an ill-conditioned matrix, then $\mathbf{A}^T\mathbf{A}$ will be a terribly ill-conditioned matrix: the issue has gotten exacerbated by the method we chose to employ. This is a nice example of the interplay between the ill-conditioning of a problem and the (possible) instability of an algorithm. There exist other, more robust algorithms, typically involving orthogonalization, as you will find out in the problem set.

Variances

For the case of the straight-line fit of section 6.5.2, after determining the estimated parameters c_0 and c_1, we proceeded to find the uncertainties in these estimates, σ_{c_0} and σ_{c_1}. You will not be surprised to hear that something analogous can be done in the present case, also. The general relation in Eq. (6.129) has the form:

$$\sigma_{c_i}^2 = \sum_{j=0}^{N-1} \left(\frac{\partial c_i}{\partial y_j} \right)^2 \sigma_j^2 \tag{6.156}$$

where we have, implicitly, assumed that the uncertainties in any two data points are uncorrelated, i.e., that the measurements are independent. To compute $\sigma_{c_i}^2$ we see that we'll need the derivative of the parameter c_i with respect to the datum y_j, i.e., $\partial c_i / \partial y_j$. To get that, in its turn, we first need to write out c_i as a function of y_j.

Multiplying Eq. (6.144) with $(\mathbf{A}^T\mathbf{A})^{-1}$ on the left, we find:

$$\mathbf{c} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b} \tag{6.157}$$

The inverse matrix is important enough that it gets its own symbol:

$$\mathbf{V} = (\mathbf{A}^T\mathbf{A})^{-1}, \qquad \mathbf{U} = \mathbf{A}^T\mathbf{A} \tag{6.158}$$

where we also went ahead and introduced a new symbol for $\mathbf{A}^T\mathbf{A}$ itself, for later use. We

can write down the relation in Eq. (6.157) for the i-th component:

$$c_i = \sum_{k=0}^{n-1} V_{ik}(\mathbf{A}^T\mathbf{b})_k = \sum_{k=0}^{n-1} V_{ik} \sum_{l=0}^{N-1} A_{lk} b_l = \sum_{k=0}^{n-1} V_{ik} \sum_{l=0}^{N-1} \frac{\phi_k(x_l)}{\sigma_l} \frac{y_l}{\sigma_l} \tag{6.159}$$

In the second equality we expressed $(\mathbf{A}^T\mathbf{b})_k$ as in Eq. (6.141), and in the third equality we wrote out A_{lk} and b_l as per their definitions in Eq. (6.140). The final expression is the result we were after, relating c_i to y_l.

Crucially, V_{ik} does not depend on y_j. This means that we can straightforwardly evaluate the needed derivative:

$$\frac{\partial c_i}{\partial y_j} = \sum_{k=0}^{n-1} V_{ik} \frac{\phi_k(x_j)}{\sigma_j^2} \tag{6.160}$$

where you'll notice that the sum over l is now gone. We now plug this result into Eq. (6.156):

$$\sigma_{c_i}^2 = \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} V_{ik} V_{il} \left(\sum_{j=0}^{N-1} \frac{\phi_k(x_j)\phi_l(x_j)}{\sigma_j^2} \right) \tag{6.161}$$

where we interchanged the order of summation and cancelled one of the σ_j^2's. Now, the term inside the parentheses can also be re-expressed:

$$\sum_{j=0}^{N-1} \frac{\phi_k(x_j)\phi_l(x_j)}{\sigma_j^2} = \sum_{j=0}^{N-1} A_{jk} A_{jl} = (\mathbf{A}^T\mathbf{A})_{kl} = U_{kl} \tag{6.162}$$

In the first equality we used the definitions in Eq. (6.140). In the second equality we replaced the sum by matrix–matrix multiplication. In the third equality we identified our new matrix from Eq. (6.158). Now we can take this result and re-introduce it into Eq. (6.161):

$$\sigma_{c_i}^2 = \sum_{k=0}^{n-1} \sum_{l=0}^{n-1} V_{ik} V_{il} U_{kl} = \sum_{l=0}^{n-1} V_{il} \sum_{k=0}^{n-1} V_{ik} U_{kl} = \sum_{l=0}^{n-1} V_{il} \delta_{il} \tag{6.163}$$

In the second equality we re-ordered the sums and terms, and in the third equality we realized that $\mathbf{V}$ is the inverse of $\mathbf{U}$, so their product is the identity. This leads us to the significant result that:

$$\sigma_{c_i}^2 = V_{ii} \tag{6.164}$$

namely, the diagonal elements of the $\mathbf{V}$ matrix are the variances of the estimates of the $\mathbf{c}$ parameters. In a problem, you will show that the off-diagonal matrix elements of $\mathbf{V}$ are the covariances between different parameters, i.e., V_{ik} is the covariance between parameters c_i and c_k. As a result, $\mathbf{V}$ is known as the *variance-covariance matrix*.

Implementation

We are now ready to implement our general linear least-squares fitting procedure in Python, see Code 6.6. We will only determine the c_k's, leaving the determination of the variances, which requires a matrix inversion, as an end-of-chapter problem.

To highlight the generality of our machinery, we decide to generate the data from a sinusoidal function; we perturb the data values using samples drawn from the standard normal distribution, conveniently provided by `numpy.random.randn()`. We ensure that we'll get the same answer every time we call our `generatedata()` function by first seeding the random-number generator, via `numpy.random.seed()`. (We will re-encounter random numbers in the following chapter, in our discussion of stochastic integration.) We also employ the function `numpy.random.randn()` to produce the "measurement" errors; we take the absolute value to ensure that all σ_j's are positive. As usual, we bundle all the data-related code lines in the same function; if you're dealing with a different set of data, simply don't call this one function.

We then have to think about how to program our basis functions, namely the $\phi_k(x)$. In the spirit of showing that our formalism is truly general, we implement two distinct sets of basis functions: (a) monomials, and (b) sinusoidal behavior. Of course, for both cases the dependence on the c_k's is linear, as per Eq. (6.136). Since the data were drawn from a sinusoidal function, it should come as no surprise that using an analogous function will do a good job capturing what's going on (modulo the "noise" that we added in by hand). In the function `phi()` we are choosing to differentiate between the two different types of theory based on which value of the size of the problem, n, was passed in. We are hard-coding only two possibilities: (a) $n = 5$ is interpreted as monomials up to fourth degree, and (b) $n = 2$ is interpreted as either 1 or $\sin(x)$. (Note how the last two options were squeezed onto one line of code.)

While the two functions we just discussed were special, applying to a given set of data or a couple of possible theories, the next function is completely general. We define a function called `normalfit()` which computes the c_k's and calculates the minimum χ^2. Its parameter list is similar to our earlier function `computecs()` from `linefit.py`; the only change is that this time we are also explicitly passing in the number of parameters, n. Note that we don't need to pass in N, since we're passing in the data itself. We immediately create the rectangular $N \times n$ matrix `A`. This is the first rectangular matrix we've seen in this book, but observe that `numpy` handles everything as smoothly as before. We iterate through the columns (of which there are n), directly implementing the definition of A_{jk} from Eq. (6.140). As usual, we employ `numpy` functionality and therefore only need a column index: the rows are iterated over implicitly, by manipulating the input data arrays. We similarly define the **b** column vector as per Eq. (6.140). Having computed **A** and **b**, we manage to both set up and solve the linear system $\mathbf{A}^T\mathbf{A}\mathbf{c} = \mathbf{A}^T\mathbf{b}$ of Eq. (6.144) in a single line. This would have been unpleasant without the convenience of using `@` to carry out matrix–matrix or matrix-vector multiplication. We call our usual routine that does Gaussian elimination with pivoting and then return the c_k's. Having produced the c_k's, we can then compute the minimum χ^2, again in one line. This line implements Eq. (6.149): in a problem, you are asked to calculate χ^2 starting from Eq. (6.137) and calling `phi()`; as

normalfit.py Code 6.6

```
from gauelim_pivot import gauelim_pivot
import numpy as np

def generatedata(N):
     np.random.seed(45379)
     dataxs = np.linspace(0,9,N)
     datays = 2 + 5*np.sin(dataxs) + 0.3*np.random.randn(N)
     datasigs = 0.2*np.abs(np.random.randn(N))
     return dataxs, datays, datasigs

def phi(n,k,x):
     if n==5:
          val = x**k
     elif n==2:
          val = 1. if k==0 else np.sin(x)
     return val

def normalfit(dataxs,datays,datasigs,n):
     N = dataxs.size
     A = np.zeros((N,n))
     for k in range(n):
          A[:,k] = phi(n,k,dataxs)/datasigs
     bs = datays/datasigs
     cs = gauelim_pivot(A.T@A, A.T@bs)
     chisq = np.sum((bs - A@cs)**2)
     return cs, chisq

if __name__ == '__main__':
     dataxs, datays, datasigs = generatedata(8)
     for n in (5, 2):
          cs, chisq = normalfit(dataxs, datays, datasigs, n)
          print(cs)
          print(chisq/(dataxs.size-cs.size)); print("")
```

you'll discover there, that is a messier task. Instead, here we take advantage of the fact that we've already computed $\mathbf{A}$ and $\mathbf{b}$.

The main program picks $N = 8$ and then tries out the two theories in turn, printing out the

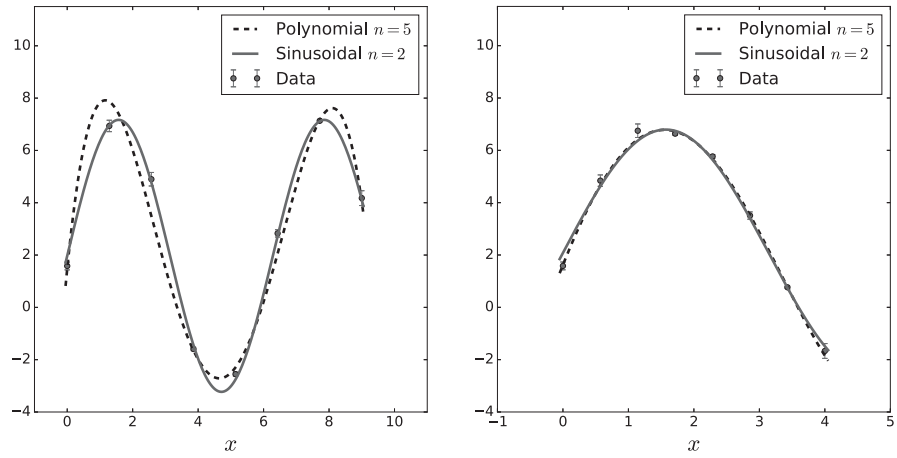


Fig. 6.13 Least-squares fitting for a polynomial and a sinusoidal theory

c_k's and minimum χ^2 each time. Running this code, we see that the sinusoidal theory did a much better job capturing the behavior of the data. This is reflected in the smaller value of the minimum χ^2. It is also clearly seen in the left panel of Fig. 6.13: despite having five parameters at its disposal, the polynomial was not able to describe the data that well. You may be thinking that this was inevitable, given that we already knew which function the data were drawn from (roughly).

That being said, you shouldn't be too quick to dismiss the accomplishments of our polynomial theory. The right panel of Fig. 6.13 shows the result of choosing the data from 0 to 4 (instead of from 0 to 9), with every other aspect of the code left unchanged. In this case, the polynomial seems to have done at least as good a job; as a matter of fact, if you compare the minimum χ^2 (per degree of freedom) for the two models, the polynomial seems to be doing *better*. This raises a host of related questions: should you trust a smaller minimum χ^2 blindly? Generally speaking, you should try to balance the magnitude of the minimum χ^2 with the number of parameters you are employing: as mentioned earlier, if you use many parameters you may end up overfitting, i.e., capturing some of the noise in the data. One of the problems asks you to study our earlier straight-line data using polynomial models of increasing degree, where this issue is thrown into sharp relief.

6.6 Project: Testing the Stefan–Boltzmann Law

As usual, this Project will discuss an application of the concepts described earlier in this chapter, for a case where a computer is necessary. Since we spent quite a bit of time discussing interpolation and less time on least-squares fitting, we take the opportunity to extend some of our earlier material to the case of nonlinear least-squares fitting. The physics problem we have chosen to study is the extraction of the exponent in the *Stefan–Boltzmann*

law for the power radiated by a black body. This is a result that preceded early quantum theory and in which the fitting to experimental data played an important role historically.

6.6.1 Beyond Linear Fitting

Our general least-squares fitting formalism, employing the normal equations, is not fully general: everything we have said so far applies to the case of Eq. (6.136), i.e., the case of linear dependence on the parameters. However, in practice you may be faced with more complicated scenarios, e.g.:

$$p(x) = c_0 e^{-c_1 x} \tag{6.165}$$

The dependence on c_1 is nonlinear, as you will discover immediately when you try to take the derivative of $p(x)$ with respect to c_1. There is a time-honored trick you could employ here: taking the natural logarithm of both sides, you are led to:

$$q(x) = d_0 + d_1 x \tag{6.166}$$

where $q(x) = \ln[p(x)]$, and the d_k's are straightforwardly related to the c_k's. Then, you could simply carry out a straight-line fit for $q(x)$ and later translate what you've learned back to the $p(x)$ model. That being said, you should keep in mind that our maximum-likelihood interpretation of the parameters that resulted from the χ^2-minimization procedure followed from the assumption that the errors were normally distributed. Manipulating your theory so that it becomes linear in the parameters will quite likely remove the Gaussianity of your measurement errors. Even so, the result of the χ^2 minimization might still be helpful to you, practically speaking.

There are many other cases of nonlinearities, which are not easy to transform away. For example, here is the *Cauchy distribution*:

$$p(x) = \frac{1}{\pi} \frac{c_1}{(x - c_0)^2 + c_1^2} \tag{6.167}$$

For this case, if you have a set of input data $(x_j, y_j \pm \sigma_j)$ and wish to determine c_0 and c_1, our earlier discussion cannot help you very much. The task at hand is to minimize the χ^2 in the general case, namely Eq. (6.120):

$$\chi^2 = \sum_{j=0}^{N-1} \left(\frac{y_j - p(x_j)}{\sigma_j} \right)^2 \tag{6.168}$$

As you may recall, for the linear case this problem was reformulated into a simple expression, $\chi^2(\mathbf{c}) = (\mathbf{b} - \mathbf{A}\mathbf{c})^T(\mathbf{b} - \mathbf{A}\mathbf{c})$, minimizing which led to a linear system of equations, $\mathbf{A}^T\mathbf{A}\mathbf{c} = \mathbf{A}^T\mathbf{b}$. For a general $p(x)$ this is no longer true, so we need to turn to general aspects of multidimensional minimization, as discussed in section 5.5.2.

For the specific problem of nonlinear χ^2 minimization, several approaches (e.g., the Gauss–Newton and Levenberg–Marquardt methods) have been developed. Instead of going over those, we will try to build your intuition by following a different tack: we will study a specific nonlinear model and explicitly carry out the derivatives of Eq. (6.121) by hand:

$$\frac{\partial \chi^2}{\partial c_k} = -2 \sum_{j=0}^{N-1} \left(\frac{y_j - p(x_j)}{\sigma_j^2} \right) \frac{\partial p(x_j)}{\partial c_k} = 0 \tag{6.169}$$

where $k = 0, 1, \ldots, n-1$. Note that this is still a general set of equations, since we haven't specified anything about $p(x)$. Thus, we will be faced with n nonlinear equations in n unknowns. To be clear, we intend to carry out these derivatives analytically, thereby converting the multidimensional minimization problem to a multidimensional root-finding problem. We will then proceed to solve the latter using the techniques we developed in section 5.4 on systems of nonlinear equations.

6.6.2 Total Power Radiated by a Black Body

We now turn to our physics theme, the total amount of radiation emitted by a black body. As you may recall, a *black body* is a perfect absorber of radiation. For reasons that will soon become clear, we will discuss in more detail than usual the history of this subject [47, 62], whose major events all took place in the second half of the nineteenth century.

Stefan's Stab in the Dark

In the early 1860s John Tyndall reported on his experiments heating a platinum wire and measuring the deflection of a galvanometer. Adolph Wüllner, in the 1875 edition of his textbook on experimental physics, summarized Tyndall's findings, providing estimates for the temperatures at which the platinum wire was observed. Thus, Tyndall's "faint red" became 525 °C and his "full white" turned into 1200 °C. The "intensity of the radiation" was observed to increase from 10.4 to 122, i.e., by a factor of 11.7. The qualitative point is that, even though the temperature increased by a factor of roughly 2, the radiation emitted increased considerably more.

In 1879 Josef Stefan published an article in which he converted (Wüllner's estimates of) Tyndall's temperatures into absolute temperatures. Stefan also noticed that by raising the ratio of the two absolute temperatures to the fourth power, one gets a value of 11.6, which is very similar to the 11.7 of the previous paragraph. Explicitly:

$$\frac{(273 + 1200)^4}{(273 + 525)^4} \approx 11.6 \tag{6.170}$$

From this, Stefan drew the general conclusion that the "heat radiation" is proportional to the fourth power of the absolute temperature.

You may be thinking that drawing a general conclusion about a power-law dependence based on only two points is stretching the limits of plausibility. You may also be thinking that Wüllner's temperatures were merely estimates. It turns out that Stefan was even luckier than that: Tyndall's measurements did *not* probe black-body radiation; repeating Tyndall's experiment today would lead to a ratio of 18.6, instead of the 11.7 following

from Tyndall's/Wüllner's numbers. This is a pretty striking example of serendipity: Stefan made the right inference, using very little (and incorrect) experimental data.

Boltzmann's Thermodynamic Derivation

In 1884 Ludwig Boltzmann, who had earlier been Stefan's doctoral student, published a paper in which he derived the Stefan law from purely thermodynamic arguments. Nowadays, we call it the *Stefan–Boltzmann law* and write it as follows:

$$I = \sigma T^4 \tag{6.171}$$

where I is the radiated energy per second per surface area, at absolute temperature T. The proportionality factor, σ, is now known as the Stefan–Boltzmann constant. This law describes the energy emitted *at all wavelengths*. It is also significant to note that, as shown by Boltzmann's derivation, this law is a *classical result*. In one of the problems you are asked to recover this result starting from Planck's formula for the black-body spectrum, thereby deriving the Stefan–Boltzmann constant, σ, in terms of fundamental constants.

Given the beauty and simplicity of Boltzmann's derivation, we now briefly go over its main steps. Suppose we have a closed system, made up of a "gas" of electromagnetic radiation. The fundamental thermodynamic relation is:

$$dU = TdS - PdV \tag{6.172}$$

where U is the internal energy, T the temperature, S the entropy, P the pressure, and V the volume. Dividing by dV at fixed T gives us:

$$\left(\frac{\partial U}{\partial V}\right)_T = T\left(\frac{\partial S}{\partial V}\right)_T - P = T\left(\frac{\partial P}{\partial T}\right)_V - P \tag{6.173}$$

where the second step followed from one of the most common *Maxwell relations*.

Maxwell's name is also relevant to the next step, since he came up with the formula for *radiation pressure* (which you will explicitly show in a problem), $P = \mathcal{E}/3$. Here $\mathcal{E} = U/V$ is the energy density of radiation (assumed to depend only on T). Plugging Maxwell's result and $U = \mathcal{E}V$ into Eq. (6.173), we get:

$$\mathcal{E} = \frac{1}{3}T\frac{d\mathcal{E}}{dT} - \frac{1}{3}\mathcal{E} \tag{6.174}$$

We wrote $d\mathcal{E}$ since $\mathcal{E}$ is a function of the temperature solely. This can be integrated to give:

$$\mathcal{E} = bT^4 \tag{6.175}$$

where b is a constant. You can immediately see the formal similarity with the Stefan–Boltzmann law of Eq. (6.171). One of the problems guides you toward relating this b constant with the Stefan–Boltzmann constant, σ.

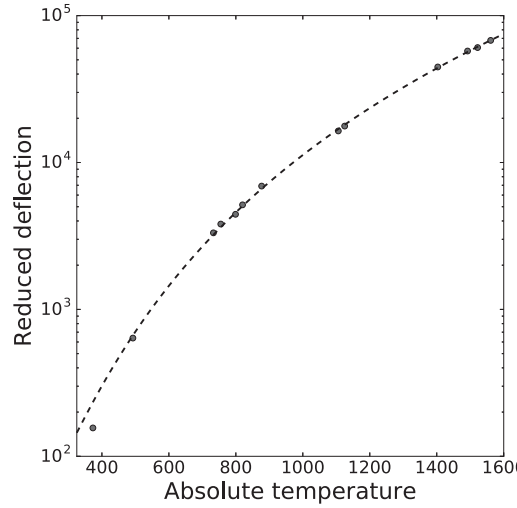


Fig. 6.14 Black-body radiation data from the 1897 paper by Lummer and Pringsheim

Lummer and Pringsheim's Experiments

In 1897, well-controlled experiments were carried out by Otto Lummer and Ernst Pringsheim [63]. In their experimental setup, Lummer and Pringsheim employed vessels coated with platinum black and surrounded by baths of water and niter; they used a bolometer to measure the radiant energy. They checked that the deflection in the galvanometer was proportional to the energy of the radiation.

Their results are shown in Fig. 6.14. Notice immediately that the y axis is labelled "reduced deflection": we don't get into details on the manipulations of the data the authors carried out. Instead, we will treat the numbers in this figure as the raw data, (x_j, y_j) for $j = 0, 1, \ldots, 12$. Another thing that immediately stands out is that the y_j's are not accompanied by measurement errors, σ_j's. This is standard in early experimental papers, which generally only made a few qualitative statements about the errors in the main text, instead. For us, this will be an opportunity to learn what to do (and how to interpret our results) when there are no input measurement errors available. We are also showing in Fig. 6.14 our least-squares fitting results pre-emptively, which we will produce in the following section. Note that we are using a logarithmic scale for our y axis and a linear scale for our x axis. You can probably already guess that on a log-log plot the points would largely end up on a straight line; this follows immediately from the Stefan–Boltzmann law, Eq. (6.171), which does look like a straight line in a log-log plot.

6.6.3 Fitting to the Lummer and Pringsheim Data

Since, as we already saw, Stefan's conjecture was based on little data which did not apply to the case in hand, the Lummer and Pringsheim work constitutes the first piece of experi-

mental evidence confirming Stefan's intuition and Boltzmann's derivation. This motivates our choice to study this set of data in some detail, trying to extract the physics ourselves.

For most of their paper, Lummer and Pringsheim *assumed* the Stefan–Boltzmann law of Eq. (6.171) was true. In other words, they assumed that the data followed a power-law with temperature raised to the fourth power. Then, they tested to see how universal the b (or σ) constant is. In what follows, we will implement a more general strategy: don't assume that T^4 gives you the energy; instead, keep the exponent general and see if the experimental measurements lead you to the Stefan–Boltzmann law.

Setting up the Equations

As noted, we will take the (x_j, y_j) points of Fig. 6.14 as our input data. It would be nice if we could assume that the data could be modelled by the theory:

$$p(x) = c_0 x^{c_1} \tag{6.176}$$

In that case, we could use the trick mentioned near Eq. (6.165), namely to take the logarithm on both sides. This would lead to a linear dependence on two parameters, in which case everything we said in section 6.5.3 would apply here. However, in the spirit of letting the data guide us it is more appropriate to study, instead, the following theory:

$$p(x) = c_0 + c_1 x^{c_2} \tag{6.177}$$

Except for the dependence on c_0, things are no longer so simple: even the c_1, which appears straightforward, is not multiplying a basis function $\phi_k(x)$, but x^{c_2}, which involves an undetermined parameter. This is a more general model than that in Eq. (6.176).

The fact that c_1 and c_2 appear in non-trivial combinations is highlighted when you evaluate the partial derivatives you'll need for Eq. (6.169):

$$\frac{\partial p(x_j)}{\partial c_0} = 1, \qquad \frac{\partial p(x_j)}{\partial c_1} = x_j^{c_2}, \qquad \frac{\partial p(x_j)}{\partial c_2} = c_1 x_j^{c_2} \ln x_j \tag{6.178}$$

Now that we have these three derivatives, we can plug them into the right-hand side of Eq. (6.169) to produce the three nonlinear coupled equations that we need to solve:

$$\begin{aligned} f_0(c_0, c_1, c_2) &= \sum_{j=0}^{N-1} \left(\frac{y_j - c_0 - c_1 x_j^{c_2}}{\sigma_j^2} \right) = 0 \\ f_1(c_0, c_1, c_2) &= \sum_{j=0}^{N-1} \left(\frac{y_j - c_0 - c_1 x_j^{c_2}}{\sigma_j^2} \right) x_j^{c_2} = 0 \\ f_2(c_0, c_1, c_2) &= \sum_{j=0}^{N-1} \left(\frac{y_j - c_0 - c_1 x_j^{c_2}}{\sigma_j^2} \right) c_1 x_j^{c_2} \ln x_j = 0 \end{aligned} \tag{6.179}$$

Even the first equation, which led to an "easy" partial derivative in Eq. (6.178), contains nonlinearities in the numerator. The other two equations are even more complicated.

You should understand that when we refer to nonlinearities, we are talking about the c_k's: this is emphasized by writing the three equations as $f_0 = f_1 = f_2 = 0$ (where all three functions are functions of c_0, c_1, and c_2), employing the notation of section 5.4 on multidimensional root-finding. Keep in mind that x_j here refers to input data: the y_j's and x_j's are known, but the c_k's are unknown. It's easy to understand that these three coupled nonlinear equations (each of which involves a sum over 13 terms) could not have been tackled straightforwardly without the use of a computer. You may be thinking: couldn't one simply define variables like S_x, S_y, and so on, as in Eq. (6.125)? The problem is that here we cannot introduce such sums that depend *only* on the input data: the c_2-dependent exponent gets in the way.

One last point: our three equations in Eq. (6.179) involve the measurement errors, σ_j's, since that's how we had set up our vanishing-derivative condition of Eq. (6.169). As you can see from Fig. 6.14, though, this time we were provided only with a set of (x_j, y_j) points. What we'll do is to take $\sigma_j = 1$ for each data pair: this actually simplifies our equations in Eq. (6.179). This assumption is equivalent to trusting each data pair the same; as you can see from Eq. (6.168), it is the *uniformity* of the variance that matters here: even if we took $\sigma_j = A$ for some fixed value A, the minimization process is not impacted. Just to warn you ahead of time, since we are dealing with a nonlinear model and no input errors, the minimum χ^2 we find from Eq. (6.168) will be nowhere near $N - n$ (which in our case is $13 - 3 = 10$).[36] That's not an issue, though, since right now we are not interested in *model selection* but in *parameter estimation*, taking the model in Eq. (6.177) as given. As usual, while there is much more to be said on the statistical concepts involved in nonlinear fitting, here we take the practical route of minimizing the χ^2, which for our case is simply minimizing the sum of the squares of the residuals.

Implementation

Since the main part of our calculation will involve solving three nonlinear equations in three unknowns, we start Code 6.7 by importing functions implementing the Jacobian matrix and Newton's method from our code `multi_newton.py` from the previous chapter. As you may recall, `jacobian()` evaluated the derivatives numerically; `multi_newton()` used Gaussian elimination to solve a linear system of equations, and terminated when the vector iterate stopped changing.

In order to avoid the use of global variables,[37] we define a function, `generatedata()`, which sets up the `numpy` arrays for the x_j's, y_j's, and σ_j's.[38] We just mentioned that for us all the σ_j's are equal to 1, but we're still keeping things general: this might help you if you were to estimate them in some other way later on: you would need to change only a single function, `generatedata()`. Similarly, if you were to study another set of data, you would only have to change this one location in the code.

[36] Of course, for a nonlinear model it's not even obvious that we are dealing with $N - n$ degrees of freedom.
[37] This is another case where object orientation would be useful.
[38] The larger y_j values seem awfully round, making us wonder if we should have trusted all data points equally.

blackbody.py Code 6.7

```
from multi_newton import jacobian, multi_newton
import numpy as np

def generatedata():
     dataxs = np.array([373.1, 492.5, 733, 755, 799, 820,
               877, 1106, 1125, 1403, 1492, 1522, 1561])
     datays = np.array([156., 638, 3320, 3810, 4440, 5150,
               6910, 16400, 17700, 44700, 57400, 60600, 67800])
     datasigs = np.ones(dataxs.size)
     return dataxs, datays, datasigs

def model(cs,x):
     return cs[0] + cs[1]*x**cs[2]

def fs(cs):
     dataxs, datays, datasigs = generatedata()
     c0, c1, c2 = cs
     resids = datays - model(cs, dataxs)
     f0 = np.sum(resids/datasigs**2)
     f1 = np.sum(dataxs**c2*resids/datasigs**2)
     numers = c1*dataxs**c2*np.log(dataxs)*resids
     f2 = np.sum(numers/datasigs**2)
     return np.array([f0,f1,f2])

def computechisq(cs):
     dataxs, datays, datasigs = generatedata()
     chisq = np.sum((datays - model(cs,dataxs))**2/datasigs**2)
     return chisq

if __name__ == '__main__':
     colds = np.array([-700, 1.26e-8, 6])
     cs = multi_newton(fs,jacobian,colds,kmax=500,tol=2.e-6)
     chisq = computechisq(cs); print(cs); print(chisq)
```

In the same spirit of producing a separate function for each task, we define a function, `model()`, to evaluate our theory, $p(x)$ from Eq. (6.177). This is called in our next function, `fs()`, which defines the f_0, f_1, and f_2 we'll need to pass to `multi_newton()` and which

it, in its turn, will pass to `jacobian()`. Since those functions had an interface which knew nothing about `dataxs` and so on, we write our `fs()` in such a way that it produces its own input data: this means that it doesn't take them in as parameters, as that would break the interface of our earlier functions.[39] Of course, it is highly inefficient to reproduce the input data each time `fs()` is called, but that is the price we are willing to pay in order to maintain interoperability with our earlier routines. (Plus, for the small-size problem we're handling here, it doesn't really matter.) The core of `fs()` sets up the three functions in Eq. (6.179); to do so in a readable manner, we first set up the residuals via calls to `model()`. As usual, that function works just as well when the input is a single number or an entire `numpy` array. One should be careful not to overinterpret the modularity of our code: while it's nice that we have `model()` at our disposal and don't need to rewrite it out each time we need it, the lines of code computing `f0`, `f1`, and `f2` are implementing Eq. (6.179), which *knows* that our model is that of Eq. (6.177). In other words, if you wish to try out a different model, you will need to modify both `model()` and `fs()`.

Finally, note that the parameters of the function `fs()` are called `cs`, since these are our unknowns. If you look at our older code in `multi_newton.py`, you should make sure not to get confused by the fact that it employs `xs` for the variables: those were the unknowns there, whereas here the x_j's are simply input values. It is quite gratifying that the implementation details of that earlier code are not important to us now: if you know how to call `multi_newton()`, i.e., its interface and the interpretation of its parameters, that is enough. You could even treat `multi_newton()` as a "black box" of your own making.

We also define another function to compute the χ^2, similarly to what we did in Code 6.5. At this stage, this is only meant to be used for the minimum χ^2, after we've solved our three-dimensional nonlinear problem (but keep reading). In the spirit of not repeating the expression for our model, we call the relevant function when we need to evalute Eq. (6.177).

The main program starts by writing down an initial guess for the c_k's. If the following point was not clear to you up to here, it should now be clear: the linear least-squares problem (say, in the case of the normal equations) was solved using linear-algebra methods, i.e., in a number of steps that was known ahead of time. In contradistinction to this, the nonlinear least-squares problem is inherently *iterative*, in the sense of us not knowing ahead of time how many iterations we'll need. Looked at another way, we will be calling the multidimensional Newton's method routine, which requires an initial guess to get going. The question then arises how to pick this initial guess; at this point, textbooks usually state general facts, e.g., that multidimensional root-finding (or minimization) is tricky business, so you may not be able to find a solution unless you start sufficiently close to it. That is certainly true, but it doesn't help us much in our practical task of deciding what to pass to Newton's method as an initial guess. You should experiment with crude guesses like `colds = np.array([1., 2, 3])` and see what happens.

A slightly more systematic way of going about things is to assume the Stefan–Boltzmann exponent is 4, *only* to get things going. In other words, you could take the last two data

[39] For your own edification, you might wish to investigate the functionality provided by `functools.partial()`.

points and assume c_0 and c_1 can be chosen to make the curve go through the points:

$$c_0 + c_1 1522^4 = 60600, \qquad c_0 + c_1 1561^4 = 67800 \tag{6.180}$$

This gives you $c_0 \approx -700$ and $c_1 \approx 1.26 \times 10^{-8}$. These values shouldn't be trusted too much, but they should be good enough to start with. We could also take the initial c_2 to be 4, but we don't want to do all the work of Newton's method for it, so we start with a guess of 6 to see what happens. It's already looking like the three parameters will have very different magnitudes.

As an aside, if you are worried that c_0 is non-zero (whereas the Stefan–Boltzmann law has no offset), don't be: we are describing the reduced data, which have been manipulated such that the energy would not go to zero if you extrapolated down to zero temperature.

The output of running this code is:

```
1 [ -2.83218318e+10 1.25998970e-08 5.94426940e+00] 1.00938366559
...
38 [ -1.51552789e+02 1.25998514e-08 3.98696831e+00] 9.70204845233e-09
[ -1.51552789e+02 1.25998514e-08 3.98696831e+00]
2203474.76723
```

A few things immediately jump out. First, the converged parameters indeed exhibit very different orders of magnitude, as was also true of the initial-guess parameters.

Second, as you may recall, Newton's method prints out the current solution vector as well as a measure of how much it has changed from the previous iteration. We see that the solution vector change is not monotonic, taking a few dozen iterations before it finally stops changing all together (rather suddenly). During those iterations, the c_0 takes a tortuous path, jumping up to huge values and then slowly dropping back down to something of order 100. All the while, the c_1 seems to change barely at all: as a result, its converged value is essentially the same as what we provided as an initial guess. While we trust that our back of the envelope calculation in Eq. (6.180) likely gave the right order of magnitude, it's hard to believe that it also just happened to give precisely the correct value.

Third, the χ^2 that is printed out at the end seems very large, but given that we are dealing with a nonlinear model with no input errors, we don't really know what values we should have been expecting for it, i.e., we don't know if it's "bad". At a qualitative level, the `fs()` we have solved are the vanishing-derivative conditions of Eq. (6.179) so, since the iterate stopped changing, we may be tempted to think that we have, indeed, succeeded in our task of minimizing χ^2.

Further Investigation

As already noted, the converged c_k's are of wildly different magnitudes. Also, while Newton's method seems to have no trouble modifying c_0 and c_2 on the way to convergence, the c_1 is essentially left untouched. Those two facts are intimately related: as you may recall, `multi_newton()` calls the function `jacobian()`, which computes a finite-difference approximation to the Jacobian. The h employed there was 10^{-4} by default: clearly, trying to

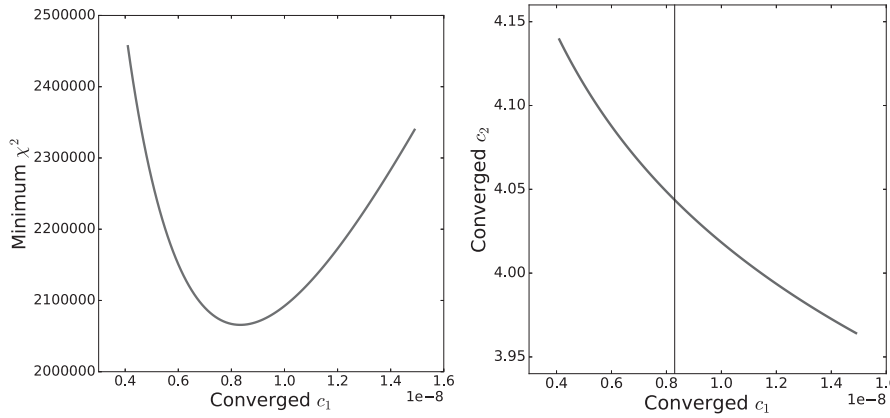


Fig. 6.15 Trying to extract the Stefan–Boltzmann exponent from the data

evaluate a quantity that has a magnitude of 10^{-8} by taking steps of 10^{-4} does not seem like a good idea. The situation, however, is even more complicated than that.

With a view to finding out more about what's going on, we decide to print out `fs(cs)` for the converged c_k's. If we have, indeed, managed to solve the three nonlinear equations, we expect to find zeros; instead, we find that f_0 was successfully solved, but the other two equations not so much. Given the very different dependences on c_1 and c_2, as well as the expectation that their magnitudes are wildly different, this is not too surprising.[40] After all, our termination criterion in Newton's method was whether or not the iterate keeps changing, not whether or not all functions have been successfully zeroed out.

Wishing to do better, we recall that c_1 was not really optimized in our run of Newton's method, staying essentially where it started. This motivates us to study the dependence on the c_1 explicitly; we'd like to vary the converged-c_1 by hand.[41] We hope that such a study will also shed some light on the question of how good or bad our minimum χ^2 was.

Since the initial guesses from Eq. (6.180) gave the expected order of magnitude, we produce a grid of c_1 values near $c_1 \approx 1.26 \times 10^{-8}$. For a given c_1 value we run the entire code, i.e., attempt to solve the three nonlinear equations, and then evaluate the χ^2 for the converged set of c_k's. The initial values of c_0 and c_2 can be kept frozen. Doing this repeatedly leads to the left panel of Fig. 6.15: we find a clear dependence of χ^2 on the value of (the converged) c_1. Even though each of these runs doesn't really change c_1 from start to finish, having many of these runs available allows us to draw a clear conclusion that the χ^2 is truly minimized for $c_1 \approx 8.3 \times 10^{-9}$. As an aside, you should realize that what we're doing here is quite wasteful: we're solving three coupled nonlinear equations using Newton's method, in an attempt to minimize the χ^2, and then minimizing *again* by hand, by repeating the entire process. This is a consequence of the specific method we are employing in order to carry out the steps in this three-parameter space.

Our manual exploration of the dependence on the value of c_1 has provided us with insight on the magnitude of our χ^2. From $c_1 \approx 1.26 \times 10^{-8}$ to $c_1 \approx 8.3 \times 10^{-9}$, the χ^2 is reduced

[40] In improved methods, one scales the relevant quantities to make them have comparable magnitude.
[41] In practice, since the initial c_1 and the converged c_1 are roughly the same, we can vary the former.

by roughly 6%. In the right panel of Fig. 6.15 we are tracking the converged value of the exponent, c_2, as we vary the c_1. The converged c_2 is always roughly 4, but the specific value we get depends (weakly) on c_1. A vertical line marks the c_1 that minimized the χ^2 in the left panel. For this case, we extract an exponent of roughly 4.04.

Having done a better job at minimizing the χ^2, you can once again print out `fs(cs)`. You'll find that all three f_i's were brought down by one or two orders of magnitude. They are still not zero, however, but that's as well as we can do for now. It's important not to take our failure personally, however. We were faced with imperfect measurements, which did not quantify the imperfections. A more detailed study of the σ_j's can plausibly help us improve the quality of our solutions. Even so, the main point still stands: minimizing the χ^2 for the Lummer–Pringsheim data led to an exponent of 4.04. Combined with the uncertainty in our estimate for this parameter, which turns out to be several orders of magnitude smaller, this is clear experimental evidence for the Stefan–Boltzmann exponent being (roughly) 4, just as intuited by Stefan and derived by Boltzmann.

6.7 Problems

1. You should compute the determinant of the Vandermonde matrix appearing in Eq. (6.17):

$$\det(\mathbf{X}) = \prod_{i>j}(x_i - x_j) \tag{6.181}$$

   The easiest way to show this is via induction. Our result shows that if $x_i \neq x_j$ for all pairs then the determinant is non-zero, and therefore the matrix is *non-singular*.
2. Produce the two plots in Fig. 6.4. Investigate the ill-conditioning by, first, evaluating the condition number of the Vandermonde matrix and, second, defining the Chebyshev points without the minus in front (i.e., in opposite order) and comparing the zoomed-in plots.
3. Implement Lagrange interpolation without the barycentric formula, i.e., the slow way, as per Eq. (6.22) and Eq. (6.19).
4. Build intuition on equispaced vs Chebyshev nodes by plotting the cardinal polynomials for $n = 10$. What do you observe regarding their magnitude?
5. Use Chebyshev points and Lagrange interpolation for (a) $f(x) = e^{\sin(20x)}$ (for $n = 15$ and $n = 60$), and (b) $f(x) = 100(x-1)^3\cos(4(x-1))e^{5(x-1)}$ (for $n = 7$ and $n = 15$).
6. Implement Lagrange interpolation for the case where the nodes are the roots of Legendre polynomials, given by `legroots.py`. Plot $n = 7$ and $n = 15$ for Runge's function.
7. We know from Eq. (6.38) that the error of the interpolant at the point x is proportional to the node-polynomial value (Eq. (6.24)) at that point. It can be formally shown that Chebyshev nodes minimize the maximum value of the node polynomial. Instead of following that route, you should plot $L(x)$ for $n = 7$ and $n = 15$ for the two choices of equidistant and Chebyshev nodes and draw the corresponding conclusions.
8. Following techniques similar to those used in section 6.2.3, you should derive the general error formula for Hermite interpolation, Eq. (6.50). Specifically, you should start in

the spirit of Eq. (6.33) and define:

$$F(x) = f(x) - p(x) - K\prod_{j=0}^{n-1}(x - x_j)^2 \tag{6.182}$$

and then follow the rest of the derivation.

9. Imagine we hadn't introduced Hermite interpolation in the main text. Try to determine an interpolating polynomial that satisfies the following conditions:

$$p(0) = 2 \qquad p(1) = -5 \qquad p'(0) = -5 \qquad p'(1) = -8 \tag{6.183}$$

Then, repeat the calculation using Hermite interpolation, Eq. (6.49).

10. The method of *piecewise-linear interpolation* is essentially an application of Lagrange interpolation in each panel, this time not to the second derivative, as in Eq. (6.67), but to the function itself. Specifically, we have:

$$s_{k-1,k}(x) = y_{k-1}\frac{x_k - x}{x_k - x_{k-1}} + y_k\frac{x - x_{k-1}}{x_k - x_{k-1}} \tag{6.184}$$

where this is to be interpreted as in Eq. (6.51). A moment's attention will highlight that this expression is nothing but Eq. (6.70) where we have dropped all the c terms Implement this in Python for Runge's function, for $n = 7$ and $n = 15$.

11. Implement the monomial three-node spline approach, as per Eq. (6.59), and compare with the output of `splines.py` for $n = 3$. Do this for many x values, leading to an $n = 3$ version of the left panel in Fig. 6.7.

12. Instead of using *natural splines* in Eq. (6.59), you could have implemented *clamped cubic splines*, in which the value of the first derivative is externally provided at the ends of the interval. Implement this and, as in the previous problem, do so for many x values, leading to the $n = 3$ version of Fig. 6.7. Take $f'(x_0) = 0.1$ and $f'(x_2) = -0.1$ as your clamped boundary conditions and discuss the effect this choice has.

13. Derive Eq. (6.70); you may choose to employ a symbolic algebra package to help with the manipulations. While you're at it, also check for the continuity of the second derivative, Eq. (6.65), for Eq. (6.70).

14. Produce the right panel in Fig. 6.7. Repeat this exercise for $f(x) = |x| - x/2 - x^2$; do you understand why Lagrange interpolation is not doing as well?

15. Explicitly use the orthogonality of sines and cosines in the continuous case to prove Eq. (6.77). Similarly, use the orthogonality of complex exponentials to show Eq. (6.80).

16. Go from the real Fourier series, Eq. (6.76), to the complex Fourier series, Eq. (6.79).

17. Explicitly show the orthogonality of the complex exponentials in the discrete case, Eq. (6.92); the trick here is to identify a geometric series. Then, use Eq. (6.92) to show the orthogonality of sines and cosines in the discrete case, Eq. (6.84); simply employ de Moivre's formula and write out the real and imaginary parts separately.[42]

[42] As the equation numbers cited here show, in the main text we went in the opposite direction i.e., started with sines and cosines and then went to complex exponentials.

18. Carry out trigonometric interpolation, similarly to `triginterp.py`, but this time for odd-n. The formulas in Eq. (6.82) and Eq. (6.85) now become:

$$\begin{aligned} a_k &= \frac{2}{n}\sum_{j=0}^{n-1} y_j \cos kx_j, \qquad k = 0, 1, \ldots, m \\ b_k &= \frac{2}{n}\sum_{j=0}^{n-1} y_j \sin kx_j, \qquad k = 1, 2, \ldots, m \\ p(x) &= \frac{1}{2}a_0 + \sum_{k=1}^{m} (a_k \cos kx + b_k \sin kx) \end{aligned} \tag{6.185}$$

    where $n = 2m + 1$. Produce one plot for $n = 7$ and another one for $n = 9$.
19. Produce plots like in Fig. 6.9, this time for $f(x) = e^{\sin x + \cos x}$ using $n = 6$ and $n = 8$.
20. Reproduce our plots in Fig. 6.9, this time using the "unshifted" discrete Fourier transform, Eq. (6.88) and Eq. (6.94). Also show the imaginary part.
21. For real input data, we can cut the number of required computations in half, because $\tilde{y}_{n-k} = \tilde{y}_k^*$ holds. Show this relationship by writing out $\tilde{y}_{n-k}$ explicitly.
22. Reproduce our plot in Fig. 6.11 by implementing $r(x)$ from Eq. (6.114).
23. This problem compares the slow DFT with the FFT.

    (a) Implement the (shifted) slow DFT of Eq. (6.97).
    (b) Carry out timing runs for the slow DFT of the previous part and the `fft()` function from `fft.py`. To make things more systematic, print out runtimes for 2^9 up to 2^{13} points, using the `default_timer()` function from the `timeit` module.

24. In our derivation from Eq. (6.103) to Eq. (6.108) we showed how the FFT approach reduces the 4×4 problem to two 2×2 ones. Write out (by hand) the analogous derivation for breaking down the 8×8 problem into two 4×4 ones.
25. This problem builds up to spectral differentiation using the FFT.

    (a) We'll need a routine implementing the inverse FFT. Thankfully, we can simply reuse the direct FFT. To see this, take the complex conjugate of Eq. (6.99): this shows that the direct DFT of $\tilde{y}_k^*$ divided by n gives you y_j^*; taking the complex conjugate of *that* then leads to y_j. Write a Python function that does this.
    (b) To evaluate derivatives, we'll start with $p(x)$ from Eq. (6.117). If you take the first derivative of $p(x)$ and then evaluate at the grid points, the cosine term (which has turned into minus sine) vanishes,[43] leading to:

$$y'_j = \frac{1}{n}\sum_{k=0}^{m-1} ik\tilde{y}_k e^{ikx_j} + \frac{1}{n}\sum_{k=m+1}^{n-1} i(k-n)\tilde{y}_k e^{i(k-n)x_j} = \frac{1}{n}\sum_{k=0}^{n-1} \tilde{y}'_k e^{ikx_j} \tag{6.186}$$

    To get to the second equality, we employed Eq. (6.83) for the x_j (in the second sum)

[43] Analogously, when taking the second derivative the cosine term does *not* vanish at the grid points. Thus, counterintuitively, the second spectral derivative is *not* equivalent to taking the first spectral derivative twice.

and also introduced a new entity, $\tilde{y}'_k$, as follows:

$$\tilde{y}'_k = \begin{cases} ik\tilde{y}_k, & 0 \le k \le m-1 \\ 0, & k = m \\ i(k-n)\tilde{y}_k, & m+1 \le k \le n-1 \end{cases} \tag{6.187}$$

Thus, to implement Eq. (6.186) you simply need the inverse DFT (or inverse FFT) of $\tilde{y}'_k$, which is defined as per Eq. (6.187). Plot the derivative for $n = 8$ and $n = 128$ and draw a conclusion about what's going on.

26. This problem deals with the covariance between the two parameters of the straight-line theory. Starting from:

$$\sigma^2_{c_0,c_1} = \sum_{j=0}^{N-1} \frac{\partial c_0}{\partial y_j} \frac{\partial c_1}{\partial y_j} \sigma_j^2 \tag{6.188}$$

show that $\sigma^2_{c_0,c_1} = -S_x/\Delta$ holds.

27. As sketched near Eq. (6.134), analytically set up the least-squares fit for the model $p(x) = c_0 + c_1 e^{-x^2}$.
28. This problem uses the straight-line data of `linefit.py`. We will employ a quadratic model, as per Eq. (6.132). You should analytically set up the problem, making it take the form of Eq. (6.133). Then, code it up in Python.
29. We now wish to generalize the approach of the previous problem to handle cubic, quartic, and so on models. Obviously, doing everything by hand is not the way to go.

    (a) You should convert `normalfit.py` to do general *polynomial fitting*.
    (b) Apply your new code to the straight-line data of `linefit.py`. Show a plot with the data, as well as the least-squares fits using polynomials of degrees 2, 3, and 4. Observe that the third degree gives essentially a straight line, increasing our confidence in the straight-line fit (whereas the fourth degree appears to be overfitting).
    (c) Plot the residuals for the three cases (of polynomials of degrees 2, 3, and 4) and interpret the results.

30. For the data in `normalfit.py`, compare the two models already contained there, with the model $p(x) = c_0 + c_1 \sin(x) + c_2 \sin(2x)$. Implement Eq. (6.164) to also compute the standard deviations. Also, code up `computechisq()` separately, using Eq. (6.137) instead of Eq. (6.149), and calling `phi()` as the need arises.
31. This problem studies the quadratic form $\beta = \mathbf{x}^T \mathbf{B} \mathbf{x}$.

    (a) Take the gradient of β, i.e., $\nabla\beta$. It is probably a good idea to break everything down in terms of components.
    (b) Now specialize to the case where $\mathbf{B}$ is a symmetric matrix.
    (c) Your previous result should be of the form $\nabla\beta = \mathbf{C}\mathbf{x}$. Now take the derivative of *this*, thereby getting the Hessian of β.

32. This problem studies the condition number for the normal-equations approach:

(a) Derive Eq. (6.155). To do this, start from:

$$(\mathbf{A} + \Delta\mathbf{A})^T(\mathbf{A} + \Delta\mathbf{A})(\mathbf{c} + \Delta\mathbf{c}) = (\mathbf{A} + \Delta\mathbf{A})^T\mathbf{b} \tag{6.189}$$

and then follow an approach similar to that of section 4.2 to show that:

$$\frac{||\Delta\mathbf{c}||}{||\mathbf{c}||} \lessapprox \kappa(\mathbf{A})^2 \frac{||\Delta\mathbf{A}||}{||\mathbf{A}||} \tag{6.190}$$

(b) Compare $\kappa(\mathbf{A}^T\mathbf{A})$ and $\kappa(\mathbf{A})^2$ for a specific ill-conditioned rectangular $\mathbf{A}$.

33. We will now see how QR decomposition can help with the least-squares problem. Assume $\mathbf{A}$ is a rectangular matrix, of dimension $N \times n$. There exists a generalization of what we saw in chapter 4, known as the *reduced QR decomposition* (or sometimes as the *thin QR decomposition*): $\mathbf{A} = \mathbf{QR}$ where $\mathbf{Q}$ is an $N \times n$ matrix with orthogonal columns and $\mathbf{R}$ is an $n \times n$ upper triangular matrix.

(a) Start from the normal equations, Eq. (6.144). Introduce the reduced QR decomposition to derive an $n \times n$ system where the left-hand side will be $\mathbf{Rc}$. Take a moment to observe that this is an upper-triangular system, which can therefore be trivially solved via back substitution.

(b) Multiply with $\mathbf{R}^T$ on the left and manipulate the equation such that $\mathbf{Q}$ no longer appears. You have now derived what are known as the *seminormal equations*. These are helpful for the case where $\mathbf{A}$ is large and sparse and you need to repeat the calculation for several different $\mathbf{b}$'s; in this scenario you'd rather avoid producing and using $\mathbf{Q}$ (which can be dense) each time.

34. This problem deals with the covariance between two parameters in the normal-equations (linear) least-squares fitting problem. Starting from:

$$\sigma^2_{c_i,c_l} = \sum_{j=0}^{N-1} \frac{\partial c_i}{\partial y_j}\frac{\partial c_l}{\partial y_j}\sigma_j^2 \tag{6.191}$$

show that $\sigma^2_{c_i,c_l} = V_{il}$ holds.

35. In the spirit of section 6.6.1, come up with analytical manipulations that will allow you to determine the parameters of the models $p(x) = 1/(c_0x + c_1)$ and $p(x) = c_0x/(x + c_1)$ using only straight-line fitting.

36. This problem addresses Maxwell's result for the radiation pressure, $P = \mathcal{E}/3$. While Maxwell carried out his derivation using only electromagnetic fields, it is easier to start from the kinetic theory of a gas of photons. For N non-interacting particles in a volume V and in thermodynamic equilibrium at temperature T, the pressure P obeys:

$$PV = \sum_j \mathbf{p}_{x,j}\mathbf{v}_{x,j} \tag{6.192}$$

where $\mathbf{p}$ is the momentum, $\mathbf{v}$ is the velocity, and x, j refers to the x component of the j-th particle. Apply this expression to photons to derive Maxwell's result.[44]

[44] If you're familiar with the covariant formulation of Maxwell electrodynamics, note that $P = \mathcal{E}/3$ can be straightforwardly derived from the vanishing of the trace of the electromagnetic stress-energy tensor.

37. In this problem we relate the energy density of black-body radiation, $\mathcal{E}$, to the energy emitted per unit area per second, I. We see that the energy in the volume element d^3r is $\mathcal{E}d^3r$. The fraction of this energy that passes through an elementary area at angle θ is $\cos\theta/(4\pi r^2)$. Thus, by integrating $\cos\theta\,\mathcal{E}\,d^3r/(4\pi r^2)$ in the upper hemisphere (ϕ from 0 to 2π, θ from 0 to $\pi/2$, and r from 0 to c), you should show that $I = c\mathcal{E}/4$.
38. Re-derive Boltzmann's Eq. (6.175) starting from the Planck distribution:

$$\mathcal{E}_\nu = \frac{8\pi h\nu^3}{c^3}\,\frac{1}{e^{h\nu/k_BT} - 1} \tag{6.193}$$

    and integrating over all (positive) frequencies ν; this gives you an expression for b. Combining this with the result of the previous problem, $I = c\mathcal{E}/4$, you can also find an expression for σ from Eq. (6.171).
39. Write Python code that produces the two plots in Fig. 6.15.
40. We now re-do the fits to the Lummer–Pringsheim data.

    (a) Use the theory of Eq. (6.176), i.e., the one with no offset. Take the logarithm on both sides, do a straight-line fit, and then translate back to the original quantities.
    (b) Again, use the theory of Eq. (6.176). This time, manually determine the c_0 for which χ^2 is minimized (using a modified version of `blackbody.py`) and then find the corresponding c_1. If the value of the c_1 doesn't agree with that from the previous part, check the variance (in the previous part).
    (c) Use the theory $p(x) = c_0x^{c_1} - c_0290^{c_1}$, which is inspired by Lummer and Pringsheim's manipulations. Again, manually determine the c_0 for which χ^2 is minimized and then find the corresponding c_1.