Chapter 11

Interpolation

So far we have derived methods for analyzing functions $f$, e.g. finding their minima and roots. Evaluating $f(\vec{x})$ at a particular $\vec{x} \in \mathbb{R}^n$ might be expensive, but a fundamental assumption of the methods we developed in previous chapters is that we can obtain $f(\vec{x})$ when we want it, regardless of $\vec{x}$.

There are many contexts when this assumption is not realistic. For instance, if we take a photograph with a digital camera, we receive an $n \times m$ grid of pixel color values sampling the continuum of light coming into a camera lens. We might think of a photograph as a continuous function from image position $(x,y)$ to color $(r,g,b)$, but in reality we only know the image value at $nm$ separated locations on the image plane. Similarly, in machine learning and statistics, often we only are given samples of a function at points where we collected data, and we must interpolate it to have values elsewhere; in a medical setting we may monitor a patient’s response to different dosages of a drug but only can predict what will happen at a dosage we have not tried explicitly.

In these cases, before we can minimize a function, find its roots, or even compute values $f(\vec{x})$ at arbitrary locations $\vec{x}$, we need a model for interpolating $f(\vec{x})$ to all of $\mathbb{R}^n$ (or some subset thereof) given a collection of samples $f(\vec{x}_i)$. Of course, techniques solving this interpolation problem are inherently approximate, since we do not know the true values of $f$, so instead we seek for the interpolated function to be smooth and serve as a “reasonable” prediction of function values.

In this chapter, we will assume that the values $f(\vec{x}_i)$ are known with complete certainty; in this case we might as well think of the problem as extending $f$ to the remainder of the domain without perturbing the value at any of the input locations. In Chapter NUMBER (WRITE ME IN 2014), we will consider the regression problem, in which the value $f(\vec{x}_i)$ is known with some uncertainty, in which case we may forgo matching $f(\vec{x}_i)$ completely in favor of making $f$ more smooth.

11.1 Interpolation in a Single Variable

Before considering the most general case, we will design methods for interpolating functions of a single variable $f : \mathbb{R} \to \mathbb{R}$. As input, we will take a set of $k$ pairs $(x_i, y_i)$ with the assumption $f(x_i) = y_i$; our job is to find $f(x)$ for $x \notin \{x_1, \ldots, x_k\}$.

Our strategy in this section and others will take inspiration from linear algebra by writing $f(x)$ in a basis. That is, the set of all possible functions $f : \mathbb{R} \to \mathbb{R}$ is far too large to work with and includes many functions that are not practical in a computational setting. Thus, we simplify the search space by forcing $f$ to be written as a linear combination of simpler building block
basis functions. This strategy is already familiar from basic calculus: The Taylor expansion writes functions in the basis of polynomials, while Fourier series use sine and cosine.

### 11.1.1 Polynomial Interpolation

Perhaps the most straightforward interpolant is to assume that \( f(x) \) is in \( \mathbb{R}[x] \), the set of polynomials. Polynomials are smooth, and it is straightforward to find a degree \( k-1 \) polynomial through \( k \) sample points.

In fact, Example 3.3 already works out the details of such an interpolation technique. As a reminder, suppose we wish to find \( f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{k-1} x^{k-1} \); here our unknowns are the values \( a_0, \ldots, a_{k-1} \). Plugging in the expression \( y_i = f(x_i) \) for each \( i \) shows that the vector \( \vec{a} \) satisfies the \( k \times k \) Vandermonde system:

\[
\begin{pmatrix}
1 & x_1 & x_1^2 & \cdots & x_1^{k-1} \\
1 & x_2 & x_2^2 & \cdots & x_2^{k-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_{k-1} & x_{k-1}^2 & \cdots & x_{k-1}^{k-1}
\end{pmatrix}
\begin{pmatrix}
a_0 \\
a_1 \\
\vdots \\
a_{k-1}
\end{pmatrix}
= 
\begin{pmatrix}
y_0 \\
y_1 \\
\vdots \\
y_{k-1}
\end{pmatrix}
\]

Thus, carrying out degree-\( k \) polynomial interpolation can be accomplished using a \( k \times k \) linear solve by applying our generic strategies from previous chapters, but in fact we can do better.

One way to think about our form for \( f(x) \) is that it is written in a basis. Just like a basis for \( \mathbb{R}^n \) is a set of \( n \) linearly-independent vectors \( \vec{v}_1, \ldots, \vec{v}_n \), here the space of polynomials of degree \( k-1 \) is written in the span of monomials \( \{1, x, x^2, \ldots, x^{k-1}\} \). It may be the most obvious basis for \( \mathbb{R}[x] \), but our current choice has few properties that make it useful for the interpolation problem. One way to see this problem is to plot the sequence of functions \( 1, x, x^2, x^3, \ldots \) for \( x \in [0,1] \); in this interval, it is easy to see that as \( k \) gets large, the functions \( x^k \) all start looking similar.

Continuing to apply our intuition from linear algebra, we may choose to write our polynomial in a basis that is more suited to the problem at hand. This time, recall that we are given \( k \) pairs \((x_1, y_1), \ldots, (x_k, y_k)\). We will use these (fixed) points to define the Lagrange interpolation basis \( \phi_1, \ldots, \phi_k \) by writing:

\[
\phi_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}
\]

Although it is not written in the basis \( 1, x, x^2, \ldots, x^{k-1} \), it is easy to see that each \( \phi_i \) is still a polynomial of degree \( k-1 \). Furthermore, the Lagrange basis has the following desirable property:

\[
\phi_i(x_\ell) = \begin{cases} 
1 & \text{when } \ell = i \\
0 & \text{otherwise.}
\end{cases}
\]

Thus, finding the unique degree \( k-1 \) polynomial fitting our \((x_\ell, y_\ell)\) pairs is easy in the Lagrange basis:

\[
f(x) = \sum_i y_i \phi_i(x)
\]

In particular, if we substitute \( x = x_j \) we find:

\[
f(x_j) = \sum_i y_i \phi_i(x_j) = y_j \text{ by our expression for } \phi_i(x_\ell) \text{ above.}
\]
Thus, in the Lagrange basis we can write a closed formula for \( f(x) \) that does not require solving the Vandermonde system. The drawback, however, is that each \( \phi_i(x) \) takes \( O(k) \) time to evaluate using the formula above for a given \( x \), so finding \( f(x) \) takes \( O(n^2) \) time; if we find the coefficients \( a_i \) from the Vandermonde system explicitly, however, the evaluation time can be reduced to \( O(n) \).

Computation time aside, the Lagrange basis has an additional numerical drawback. Notice that the denominator is the product of a number of terms. If the \( x_i \)'s are close together, then the product may include many terms close to zero, so we are dividing by a potentially small number. As we have seen this operation can create numerical issues that we wish to avoid.

One basis for polynomials of degree \( k - 1 \) that attempts to compromise between the numerical quality of the monomials and the efficiency of the Lagrange basis is the Newton basis, defined as follows:

\[
\psi_i(x) = \prod_{j=1}^{i-1} (x - x_j)
\]

We define \( \psi_1(x) \equiv 1 \). Notice that \( \psi_i(x) \) is a degree \( i - 1 \) polynomial. By definition of \( \psi_i \), it is clear that \( \psi_i(x_\ell) = 0 \) for all \( \ell < i \). If we wish to write \( f(x) = \sum_i c_i \psi_i(x) \) and write out this observation more explicitly, we find:

\[
\begin{align*}
\psi_1(x_1) & = 0 \\
\psi_1(x_2) & = 0 \\
\psi_1(x_3) & = 0 \\
\vdots & \vdots \\
\psi_1(x_k) & = 0 \\
\psi_2(x_2) & = \prod_{j=1}^{2} (x - x_j) = 0 \\
\psi_2(x_3) & = \prod_{j=1}^{2} (x - x_j) = 0 \\
\vdots & \vdots \\
\psi_2(x_k) & = \prod_{j=1}^{2} (x - x_j) = 0 \\
\psi_3(x_3) & = \prod_{j=1}^{3} (x - x_j) = 0 \\
\vdots & \vdots \\
\psi_3(x_k) & = \prod_{j=1}^{3} (x - x_j) = 0 \\
\psi_4(x_4) & = \prod_{j=1}^{4} (x - x_j) = 0 \\
\vdots & \vdots \\
\psi_4(x_k) & = \prod_{j=1}^{4} (x - x_j) = 0 \\
\vdots & \vdots \\
\psi_k(x_k) & = \prod_{j=1}^{k} (x - x_j) = 0
\end{align*}
\]

In other words, we can solve the following lower triangular system for \( \vec{c} \):

\[
\begin{pmatrix}
\psi_1(x_1) & 0 & 0 & \ldots & 0 \\
\psi_1(x_2) & \psi_2(x_2) & 0 & \ldots & 0 \\
\psi_1(x_3) & \psi_2(x_3) & \psi_3(x_3) & \ldots & 0 \\
\vdots & \vdots & \vdots & \ldots & \vdots \\
\psi_1(x_k) & \psi_2(x_k) & \psi_3(x_k) & \ldots & \psi_k(x_k)
\end{pmatrix}
\begin{pmatrix}
c_1 \\
c_2 \\
c_3 \\
\vdots \\
c_k
\end{pmatrix}
= 
\begin{pmatrix}
y_1 \\
y_2 \\
y_3 \\
\vdots \\
y_k
\end{pmatrix}
\]

This system can be solved in \( O(n^2) \) time using forward substitution, rather than the \( O(n^3) \) time needed to solve the Vandermonde system.

We now have three strategies of interpolating \( k \) data points using a degree \( k - 1 \) polynomial by writing it in the monomial, Lagrange, and Newton bases. All three represent different compromises between numerical quality and speed. An important property, however, is that the resulting interpolated function \( f(x) \) is the same in each case. More explicitly, there is exactly one polynomial of degree \( k - 1 \) going through a set of \( k \) points, so since all our interpolants are degree \( k - 1 \) they must have the same output.

### 11.1.2 Alternative Bases

Although polynomial functions are particularly amenable to mathematical analysis, there is no fundamental reason why our interpolation basis cannot consist of different types of functions. For example, a crowning result of Fourier analysis implies that a large class of functions are well-approximated by sums of trigonometric functions \( \cos(kx) \) and \( \sin(kx) \) for \( k \in \mathbb{N} \). A construction
like the Vandermonde system still applies in this case, and in fact the Fast Fourier Transform algorithm (which merits a larger discussion) shows how to carry out such an interpolation even faster.

A smaller extension of the development in §11.1.1 is to rational functions of the form:

$$f(x) = \frac{p_0 + p_1 x + p_2 x^2 + \cdots + p_m x^m}{q_0 + q_1 x + q_2 x^2 + \cdots + q_n x^n}$$

Notice that if we are given $k$ pairs $(x_i, y_i)$, then we will need $m + n + 1 = k$ for this function to be well-defined. One additional degree of freedom must be fixed to account for the fact that the same rational function can be expressed multiple ways by identical scaling of the numerator and the denominator.

Rational functions can have asymptotes and other patterns not achievable using only polynomials, so they can be desirable interpolants for functions that change quickly or have poles. In fact, once $m$ and $n$ are fixed, the coefficients $p_i$ and $q_i$ still can be found using linear techniques by multiplying both sides by the denominator:

$$y_i (q_0 + q_1 x_i + q_2 x_i^2 + \cdots + q_n x_i^n) = p_0 + p_1 x_i + p_2 x_i^2 + \cdots + p_m x_i^m$$

Again, the unknowns in this expression are the $p$’s and $q$’s.

The flexibility of rational functions, however, can cause some issues. For instance, consider the following example:

**Example 11.1** (Failure of rational interpolation, Bulirsch-Stoer §2.2). Suppose we wish to find $f(x)$ with the following data points: $(0, 1)$, $(1, 2)$, $(2, 2)$. We could choose $m = n = 1$. Then, our linear conditions become:

$$q_0 = p_0$$
$$2(q_0 + q_1) = p_0 + p_1$$
$$2(q_0 + 2q_1) = p_0 + 2p_1$$

One nontrivial solution to this system is:

$$p_0 = 0$$
$$p_1 = 2$$
$$q_0 = 0$$
$$q_1 = 1$$

This implies the following form for $f(x)$:

$$f(x) = \frac{2x}{x}$$

This function has a degeneracy at $x = 0$, and in fact canceling the $x$ in the numerator and denominator does not yield $f(0) = 1$ as we might desire.

This example illustrates a larger phenomenon. Our linear system for finding the $p$’s and $q$’s can run into issues when the resulting denominator $\sum d p_r x^r$ has a root at any of the fixed $x_i$’s. It can be shown that when this is the case, no rational function exists with the fixed choice of $m$ and $n$ interpolating the given values. A typical partial resolution in this case is presented in (CITE), which increments $m$ and $n$ alternatingly until a nontrivial solution exists. From a practical standpoint, however, the specialized nature of these methods is a good indicator that alternative interpolation strategies may be preferable when the basic rational methods fail.
11.1.3 Piecewise Interpolation

So far, we have constructed our interpolation strategies by combining simple functions on all of \( \mathbb{R} \). When the number \( k \) of data points becomes high, however, many degeneracies become apparent. For example, Figure NUMBER shows examples in which fitting high-degree polynomials to input data can yield unexpected results. Furthermore, Figure NUMBER illustrates how these strategies are nonlocal, meaning that changing any single value \( y_i \) in the input data can change the behavior of \( f \) for all \( x \), even those that are far away from the corresponding \( x_i \). Somehow this property is unrealistic: We expect only the input data near a given \( x \) to affect the value of \( f(x) \), especially when there is a large cloud of input points.

For these reasons, when we design a set of basis functions \( \phi_1, \ldots, \phi_k \), a desirable property is not only that they are easy to work with but also that they have compact support:

**Definition 11.1 (Compact support).** A function \( g(x) \) has compact support if there exists \( C \in \mathbb{R} \) such that \( g(x) = 0 \) for any \( x \) with \( |x| > C \).

That is, compactly supported functions only have a finite range of points in which they can take nonzero values.

A common strategy for constructing interpolating bases with compact support is to do so in a piecewise fashion. In particular, much of the literature on computer graphics depends on the construction of piecewise polynomials, which are defined by breaking \( \mathbb{R} \) into a set of intervals and writing a different polynomial in each interval. To do so, we will order our data points so that \( x_1 < x_2 < \cdots < x_k \). Then, two simple examples of piecewise interpolants are the following:

- **Piecewise constant (Figure NUMBER):** For a given \( x \), find the data point \( x_i \) minimizing \( |x - x_i| \) and define \( f(x) = y_i \).

- **Piecewise linear (Figure NUMBER):** If \( x < x_1 \) take \( f(x) = y_1 \), and if \( x > x_k \) take \( f(x) = y_k \). Otherwise, find an interval with \( x \in [x_i, x_{i+1}] \) and define

\[
    f(x) = y_{i+1} \cdot \frac{x - x_i}{x_{i+1} - x_i} + y_i \cdot \left(1 - \frac{x - x_i}{x_{i+1} - x_i}\right).
\]

More generally, we can write a different polynomial in each interval \([x_i, x_{i+1}]\). Notice our pattern so far: Piecewise constant polynomials are discontinuous, while piecewise linear functions are continuous. It is easy to see that piecewise quadratics can be \( C^1 \), piecewise cubics can be \( C^2 \), and so on. This increased continuity and differentiability occurs even though each \( y_i \) has local support; this theory is worked out in detail in constructing “splines,” or curves interpolating between points given function values and tangents.

This increased continuity, however, has its own drawbacks. With each additional degree of differentiability, we put a stronger smoothness assumption on \( f \). This assumption can be unrealistic: Many physical phenomena truly are noisy or discontinuous, and this increased smoothness can negatively affect interpolatory results. One domain in which this effect is particularly clear is when interpolation is used in conjunction with physics simulation tools. Simulating turbulent fluid flows with oversmoothed functions can remove discontinuous phenomena like shock waves that are desirable as output.
These issues aside, piecewise polynomials still can be written as linear combinations of basis functions. For instance, the following functions serve as a basis for the piecewise constant functions:

$$\phi_i(x) = \begin{cases} 1 & \text{when } \frac{x_{i-1} + x_i}{2} \leq x < \frac{x_i + x_{i+1}}{2} \\ 0 & \text{otherwise} \end{cases}$$

This basis simply puts the constant 1 near $x_i$ and 0 elsewhere; the piecewise constant interpolation of a set of points $(x_i, y_i)$ is written as $f(x) = \sum_i y_i \phi_i(x)$. Similarly, the so-called “hat” basis shown in Figure NUMBER spans the set of piecewise linear functions with sharp edges at our data points $x_i$:

$$\psi_i(x) = \begin{cases} \frac{x-x_{i-1}}{x_i-x_{i-1}} & \text{when } x_{i-1} < x \leq x_i \\ \frac{x_i+1-x}{x_i+1-x_{i+1}} & \text{when } x_i < x \leq x_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

Once again, by construction the piecewise linear interpolation of the given data points is $f(x) = \sum_i y_i \psi_i(x)$.

11.1.4 Gaussian Processes and Kriging

Not covered in CS 205A, Fall 2013.

11.2 Multivariable Interpolation

Many extensions of the strategies above exist for interpolating a function given datapoints $(\vec{x}_i, y_i)$ where $\vec{x}_i \in \mathbb{R}^n$ now can be multidimensional. Strategies for interpolation in this case, however, are not quite as clear, however, because it is less obvious to partition $\mathbb{R}^n$ into a small number of regions around the $x_i$. For this reason, a common pattern is to interpolate using relatively low-order functions, that is, to prefer simplistic and efficient interpolation strategies over ones that output $C^\infty$ functions.

If all we are given is the set of inputs and outputs $(\vec{x}_i, y_i)$, then one piecewise constant strategy for interpolation is to use nearest-neighbor interpolation. In this case $f(\vec{x})$ simply takes the value $y_i$ corresponding to $\vec{x}_i$ minimizing $\|\vec{x} - \vec{x}_i\|_2$; simple implementations iterate over all $i$ to find this value, although data structures like k-d trees can find nearest neighbors more quickly. Just as piecewise constant interpolations divided $\mathbb{R}$ into intervals about the data points $x_i$, the nearest-neighbor strategy divides $\mathbb{R}^n$ into a set of Voronoi cells:

**Definition 11.2 (Voronoi cell).** Given a set of points $S = \{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_k\} \subseteq \mathbb{R}^n$, the Voronoi cell corresponding to a specific $\vec{x}_i$ is the set $V_i \equiv \{\vec{x}: \|\vec{x} - \vec{x}_i\|_2 < \|\vec{x} - \vec{x}_j\|_2 \text{ for all } j \neq i\}$. That is, it is the set of points closer to $\vec{x}_i$ than to any other $\vec{x}_j$ in $S$.

Figure NUMBER shows an example of the Voronoi cells about a set of data points in $\mathbb{R}^2$. These cells have many favorable properties; for example, they are convex polygons and are localized about each $\vec{x}_i$. In fact, the connectivity of Voronoi cells is a well-studied problem in computational geometry leading to the construction of the celebrated Delaunay triangulation.

There are many options for continuous interpolation of functions on $\mathbb{R}^n$, each with its own advantages and disadvantages. If we wish to extend our nearest-neighbor strategy above, for example, we could compute multiple nearest neighbors of $\vec{x}$ and interpolate $f(\vec{x})$ based on $\|\vec{x} - \vec{x}_i\|_2 < \|\vec{x} - \vec{x}_j\|_2$ for all $j \neq i$. 


\( \vec{x}_i \) for each nearest neighbor \( \vec{x}_i \). Certain “ \( k \)-nearest neighbor” data structures can accelerate queries where you want to find multiple points in a dataset closest to a given \( \vec{x} \).

Another strategy appearing frequently in the computer graphics literature is barycentric interpolation. Suppose we have exactly \( n + 1 \) sample points \((\vec{x}_1, y_1), \ldots, (\vec{x}_{n+1}, y_{n+1})\), where \( \vec{x}_i \in \mathbb{R}^n \), and as always we wish to interpolate the \( y \) values to all of \( \mathbb{R}^n \); for example, on the plane we would be given three values associated with the vertices of a triangle. Any point \( \vec{x} \in \mathbb{R}^n \) can be written uniquely as a linear combination \( \vec{x} = \sum_{i=1}^{n+1} a_i \vec{x}_i \) with an additional constraint that \( \sum_i a_i = 1 \); in other words, we write \( \vec{x} \) as a weighted average of the points \( \vec{x}_i \). Barycentric interpolation in this case simply writes \( f(\vec{x}) = \sum_i a_i(\vec{x})y_i \).

On the plane \( \mathbb{R}^2 \), barycentric interpolation has a straightforward geometric interpolation involving triangle areas, illustrated in Figure NUMBER. Furthermore, it is easy to check that the resulting interpolated function \( f(\vec{x}) \) is affine, meaning it can be written \( f(\vec{x}) = c + \vec{d} \cdot \vec{x} \) for some \( c \in \mathbb{R} \) and \( \vec{d} \in \mathbb{R}^n \).

In general, the system of equations we wish to solve for barycentric interpolation at some \( \vec{x} \in \mathbb{R}^n \) is:

\[
\sum_i a_i \vec{x}_i = \vec{x} \\
\sum_i a_i = 1
\]

In the absence of degeneracies, this system for \( \vec{a} \) is invertible when there are \( n + 1 \) points \( \vec{x}_i \). In the presence of more \( \vec{x}_i \)'s, however, the system for \( \vec{a} \) becomes underdetermined. This means that there are multiple ways of writing a given \( \vec{x} \) as a weighted average of the \( \vec{x}_i \)'s.

One resolution of this issue is to add more conditions on the vector of averaging weights \( \vec{a} \). This strategy results in generalized barycentric coordinates, a topic of research in modern mathematics and engineering. Typical constraints on \( \vec{a} \) ask that it is smooth as a function on \( \mathbb{R}^n \) and nonnegative on the interior of the set of \( \vec{x}_i \)'s when these points define a polygon or polyhedron. Figure NUMBER shows an example of generalized barycentric coordinates computed from data points on a polygon with more than \( n + 1 \) points.

An alternative resolution of the underdetermined problem for barycentric coordinates relates to the idea of using piecewise functions for interpolation; we will restrict our discussion here to \( \vec{x}_i \in \mathbb{R}^2 \) for simplicity, although the extensions to higher dimensions are relatively obvious. Many times, we are given not only the set of points \( \vec{x}_i \) but also a decomposition of the domain we care about (in this case some subset of \( \mathbb{R}^2 \)) into \( n + 1 \)-dimensional objects using those points as vertices. For example, Figure NUMBER shows such a tessellation of a part of \( \mathbb{R}^2 \) into triangles. Interpolation in this case is straightforward: the interior of each triangle is interpolated using barycentric coordinates.

**Example 11.2** (Shading). In computer graphics, one of the most common representations of a shape is as a set of triangles in a mesh. In the per-vertex shading model, one color is computed for each vertex on a mesh. Then to render the image to the screen, those per-vertex values are interpolated using barycentric interpolation to the interiors of the triangles. Similar strategies are used for texturing and other common tasks. Figure NUMBER shows an example of this simple shading model. As an aside, one issue specific to computer graphics is the interplay between perspective transformations and interpolation strategies. Barycentric interpolation of color on a 3D surface and then projecting that color to the image plane is not the same as projecting triangles to the image plane and subsequently interpolating colors to the interior of the triangle; thus algorithms in this domain must apply perspective correction to account for this mistake.
Given a set of points in \( \mathbb{R}^2 \), the problem of triangulation is far from trivial, and algorithms for doing this sort of computation often extend poorly to \( \mathbb{R}^n \). Thus, in higher dimensions nearest-neighbor or regression strategies become preferable (see Chapter NUMBER).

Barycentric interpolation leads to a generalization of the piecewise linear hat functions from §11.1.3 illustrated in Figure NUMBER. Recall that our interpolatory output is determined completely by the values \( y_i \) at the vertices of the triangles. In fact, we can think of \( f(\vec{x}) \) as a linear combination \( \sum y_i \phi_i(\vec{x}) \), where each \( \phi_i(\vec{x}) \) is the piecewise barycentric function obtained by putting a 1 on \( \vec{x}_i \) and 0 everywhere else, as in Figure NUMBER. These triangular hat functions form the basis of the “first-order finite elements method,” which we will explore in future chapters; specialized constructions using higher-order polynomials are known as “higher-order elements” can be used to guarantee differentiability along triangle edges.

An alternative and equally important decomposition of the domain of \( f \) occurs when the points \( \vec{x}_i \) occur on a regular grid in \( \mathbb{R}^n \). The following examples illustrate situations when this is the case:

**Example 11.3** (Image processing). As mentioned in the introduction, a typical digital photograph is represented as an \( m \times n \) grid of red, green, and blue color intensities. We can think of these values as living on a lattice in \( \mathbb{Z} \times \mathbb{Z} \). Suppose we wish to rotate the image by an angle that is not a multiple of 90°, however. Then, as illustrated in Figure NUMBER, we must look up image values at potentially non-integer positions, requiring the interpolation of colors to \( \mathbb{R} \times \mathbb{R} \).

**Example 11.4** (Medical imaging). The typical output of a magnetic resonance imaging (MRI) device is a \( m \times n \times p \) grid of values representing the density of tissue at different points; theoretically the typical model for this function is as \( f: \mathbb{R}^3 \rightarrow \mathbb{R} \). We can extract the outer surface of a particular organ, showed in Figure NUMBER, by finding the level set \( \{ \vec{x} : f(\vec{x}) = c \} \) for some \( c \). Finding this level set requires us to extend \( f \) to the entire voxel grid to find exactly where it crosses \( c \).

Grid-based interpolation strategies typically apply the one-dimensional formulae from §11.1.3 one dimension at a time. For example, *bilinear* interpolation schemes in \( \mathbb{R}^2 \) linearly interpolate one dimension at a time to obtain the output value:

**Example 11.5** (Bilinear interpolation). Suppose \( f \) takes on the following values:

- \( f(0, 0) = 1 \)
- \( f(0, 1) = -3 \)
- \( f(1, 0) = 5 \)
- \( f(1, 1) = -11 \)

and that in between \( f \) is obtained by bilinear interpolation. To find \( f(\frac{1}{4}, \frac{1}{2}) \), we first interpolate in \( x_1 \) to find:

\[
\begin{align*}
    f(\frac{1}{4}, 0) &= \frac{3}{4} f(0, 0) + \frac{1}{4} f(1, 0) = 2 \\
    f(\frac{1}{4}, 1) &= \frac{3}{4} f(0, 1) + \frac{1}{4} f(1, 1) = -5
\end{align*}
\]
Next, we interpolate in $x_2$:

$$f\left(\frac{1}{4}, \frac{1}{2}\right) = \frac{1}{2} f\left(\frac{1}{4}, 0\right) + \frac{1}{2} f\left(\frac{1}{4}, 1\right) = -\frac{3}{2}$$

An important property of bilinear interpolation is that we receive the same output interpolating first in $x_2$ and second in $x_1$.

Higher-order methods like bicubic and Lanczos interpolation once again use more polynomial terms but are slower to compute. In particular, in the case of interpolating images, bicubic strategies require more data points than the square of function values closest to a point $\vec{x}$; this additional expense can slow down graphical tools for which every lookup in memory incurs additional computation time.

11.3 Theory of Interpolation

So far our treatment of interpolation has been fairly heuristic. While relying on our intuition for what a “reasonable” interpolation for a set of function values for the most part is an acceptable strategy, subtle issues can arise with different interpolation methods that are important to acknowledge.

11.3.1 Linear Algebra of Functions

We began our discussion by posing assorted interpolation strategies as different bases for the set of functions $f : \mathbb{R} \to \mathbb{R}$. This analogy of to vector spaces extends to a complete geometric theory of functions, and in fact early work in the field of functional analysis essentially extends the geometry of $\mathbb{R}^n$ to sets of functions. Here we will discuss functions of one variable, although many aspects of the extension to more general functions are easy to carry out.

Just as we can define notions of span and linear combination for functions, for fixed $a, b \in \mathbb{R}$ we can define an inner product of functions $f(x)$ and $g(x)$ as follows:

$$\langle f, g \rangle \equiv \int_a^b f(x)g(x) \, dx.$$

Just as the $A$-inner product of vectors helped us derive the conjugate gradients algorithm and had much in common with the dot product, the functional inner product can be used to define linear algebra methods for dealing with spaces of functions and understanding their span. We also define a norm of a function to be $\|f\| \equiv \sqrt{\langle f, f \rangle}$.

**Example 11.6.** Function inner product Take $p_n(x) = x^n$ to be the $n$-th monomial. Then, for $a = 0$ and $b = 1$ we have:

$$\langle p_n, p_m \rangle = \int_0^1 x^n \cdot x^m \, dx = \int_0^1 x^{n+m} \, dx = \frac{1}{n+m+1}$$
Notice that this shows:
\[
\left\langle \frac{p_n}{\|p_n\|}, \frac{p_m}{\|p_m\|} \right\rangle = \frac{\langle p_n, p_m \rangle}{\|p_n\| \|p_m\|} = \frac{\sqrt{(2n+1)(2m+1)}}{n + m + 1}
\]
This value is approximately 1 when \( n \approx m \) but \( n \neq m \), substantiating our earlier claim that the monomials “overlap” considerably on \([0, 1]\).

Given this inner product, we can apply the Gram-Schmidt algorithm to find an orthonormal basis for the set of polynomials. If we take \( a = -1 \) and \( b = 1 \), we get the Legendre polynomials, plotted in Figure NUMBER:

\[
\begin{align*}
P_0(x) &= 1 \\
P_1(x) &= x \\
P_2(x) &= \frac{1}{2}(3x^2 - 1) \\
P_3(x) &= \frac{1}{2}(5x^3 - 3x) \\
P_4(x) &= \frac{1}{8}(35x^4 - 30x^2 + 3) \\
&\vdots \\
\end{align*}
\]

These polynomials have many useful properties thanks to their orthogonality. For example, suppose we wish to approximate \( f(x) \) with a sum \( \sum_i a_i P_i(x) \). If we wish to minimize \( \|f - \sum_i a_i P_i\| \) in the functional norm, this is a least squares problem! By orthogonality of the Legendre basis for \( \mathbb{R}[x] \), a simple extension of our methods for projection shows:

\[
a_i = \frac{\langle f, P_i \rangle}{\langle P_i, P_i \rangle}
\]

Thus, approximating \( f \) using polynomials can be accomplished simply by integrating \( f \) against the members of the Legendre basis; in the next chapter we will learn how this integral might be carried out approximately.

Given a positive function \( w(x) \), We can define a more general inner product \( \langle \cdot, \cdot \rangle_w \) by writing

\[
\langle f, g \rangle_w = \int_a^b w(x)f(x)g(x) \, dx.
\]

If we take \( w(x) = \frac{1}{\sqrt{1-x^2}} \) with \( a = -1 \) and \( b = 1 \), then applying Gram-Schmidt yields the Chebyshev polynomials:

\[
\begin{align*}
T_0(x) &= 1 \\
T_1(x) &= x \\
T_2(x) &= 2x^2 - 1 \\
T_3(x) &= 4x^3 - 3x \\
T_4(x) &= 8x^4 - 8x^2 + 1 \\
&\vdots \\
\end{align*}
\]
In fact, a surprising identity holds for these polynomials:
\[ T_k(x) = \cos(k \arccos(x)). \]

This formula can be checked by explicitly checking it for \( T_0 \) and \( T_1 \), and then inductively applying the observation:
\[
T_{k+1}(x) = \cos((k + 1) \arccos(x)) \\
= 2x \cos(k \arccos(x)) - \cos((k - 1) \arccos(x)) \text{ by the identity} \\
\cos((k + 1)\theta) = 2 \cos(k\theta) \cos(\theta) - \cos((k - 1)\theta) \\
= 2xT_k(x) - T_{k-1}(x)
\]

This “three-term recurrence” formula also gives an easy way to generate the Chebyshev polynomials.

As illustrated in Figure NUMBER, thanks to the trigonometric formula for the Chebyshev polynomials it is easy to see that the minima and maxima of \( T_k \) oscillate between +1 and -1. Furthermore, these extrema are located at \( \cos(i\pi/k) \) (the so-called “Chebyshev points”) for \( i \) from 0 to \( k \); this nice distribution of extrema avoids oscillatory phenomena like that shown in Figure NUMBER when using a finite number of polynomial terms to approximate a function. In fact, more technical treatments of polynomial interpolation recommend placing \( x_i \)’s for interpolation near Chebyshev points to obtain smooth output.

### 11.3.2 Approximation via Piecewise Polynomials

Suppose we wish to approximate a function \( f(x) \) with a polynomial of degree \( n \) on an interval \([a, b]\). Define \( \Delta x \) to be the spacing \( b - a \). One measure of error of an approximation is as a function of \( \Delta x \), which should vanish as \( \Delta x \to 0 \). Then, if we approximate \( f \) with piecewise polynomials, this type of analysis tells us how far apart we should space the polynomials to achieve a desired level of approximation.

For example, suppose we approximate \( f \) with a constant \( c = f(\frac{a+b}{2}) \), as in piecewise constant interpolation. If we assume \( |f'(x)| < M \) for all \( x \in [a, b] \), we have:
\[
\max_{x\in[a,b]} |f(x) - c| \leq \Delta x \max_{x\in[a,b]} M \text{ by the mean value theorem} \\
\leq M\Delta x
\]

Thus, we expect \( O(\Delta x) \) error when using piecewise constant interpolation.

Suppose instead we approximate \( f \) using piecewise linear interpolation, that is, by taking
\[
\tilde{f}(x) = \frac{b-x}{b-a} f(a) + \frac{x-a}{b-a} f(b).
\]

By the mean value theorem, we know \( \tilde{f}'(x) = f'(\theta) \) for some \( \theta \in [a, b] \). Writing the Taylor expansion about \( \theta \) shows \( f(x) = f(\theta) + f'(\theta)(x - \theta) + O(\Delta x^2) \) on \([a, b]\), while we can rewrite our linear approximation as \( \tilde{f}(x) = f(\theta) + f'(\theta)(x - \theta) \). Thus, subtracting these two expressions shows that the approximation error of \( f \) decreases to \( O(\Delta x^2) \). It is not difficult to predict that approximation with a degree \( n \) polynomial makes \( O(\Delta x^{n+1}) \) error, although in practice the quadratic convergence of piecewise linear approximations suffices for most applications.
11.4 Problems

Ideas:

- Horner’s method for evaluating polynomials
- Recursive strategy for Newton polynomial coefficients.
- Splines, deCasteljeau
- Check triangle area interpolation of barycentric interpolation