A convenient view of interpolation
for CS514, Spring'98

The basic information about a function (or, more generally an element of a vector space) are *numbers* (i.e., scalars), like function values, value of a derivative, the definite integral of the function over some interval, the coefficients in some expansion (Taylor, Fourier, etc) for the function. Maps that associate a function with a number are called **functional**s. E.g.,

$$\delta_t : f \mapsto f(t), \quad \delta_t D^2 : f \mapsto f''(t), \quad \int_a^b : f \mapsto \int_a^b f(t)\,\mathrm{d}t$$

are functionals. All these specific examples are, in addition, **linear**, meaning that, with λ denoting any one of them,

$$\lambda(\alpha f + \beta g) = \alpha(\lambda f) + \beta(\lambda g)$$

for any two (applicable) functions f and g, and any two scalars α and β. An example of a **non**linear functional is

$$f \mapsto \|f\|_{\infty,[a,b]}.$$

If we have n such *linear* pieces of information about a function f, then we can visualize that as the value at f of a linear map of the form

$$\Lambda f := (\lambda_1 f, \lambda_2 f, \ldots, \lambda_n f).$$

We call such a linear map a **data map** since it extracts numerical data or information about f. (Such a map is also called an **analysis map**.) We are used to visualizing n-vectors as one-column matrices (so-called '*column vectors*'), hence would visualize Λ as a matrix with the n *rows* $\lambda_1, \ldots, \lambda_n$, i.e., write, in `MATLAB`-like notation,

$$\Lambda = [\lambda_1; \lambda_2; \ldots; \lambda_n].$$

This is entirely in line with one of the two standard views of matrix multiplication: the ith entry of the vector $A\mathbf{c}$ is the scalar product of the ith row of A with $\mathbf{c}$, i.e.,

$$(A\mathbf{c})(i) = \langle A(i,:), \mathbf{c}\rangle = \sum_j A(i,j)c_j.$$

We use such a data map Λ to extract numerical information from a function. In **interpolation**, we try to go the other way: we assume that we know the n-vector $\mathbf{b}$ and want to construct from this a function f for which $\Lambda f = \mathbf{b}$. Of course, this can usually be done in infinitely many ways. We single out a particular way by choosing an n-dimensional subspace (like Π_{n-1}) from which our interpolating f is to be taken. Numerically, this means that we choose some basis $(v_1, \ldots, v_n)$ for that space and then try to choose the n-vector $\mathbf{c}$ so that $\Lambda f = \mathbf{b}$ for the function $f = \sum_{j=1}^n c_j v_j$.

Now note that $\sum_{j=1}^n c_j v_j$ depends linearly on $\mathbf{c}$, i.e.,

$$V : \mathbf{c} \mapsto \sum_{j=1}^n c_j v_j$$

is a linear map. In fact, it is suggestive to write this linear map, more explicitly,

$$[v_1, \ldots, v_n]\mathbf{c} := \sum_{j=1}^n c_j v_j$$

and call it the **column map with columns** $v_1, \ldots, v_n$ since the role of the v_j in this map is entirely analogous to that of the columns $A(:,j)$ of a matrix A when we form the product of the matrix with the vector $\mathbf{c}$:

$$A\mathbf{c} = A(:,1)c_1 + A(:,2)c_2 + \cdots + A(:,n)c_n.$$

Such a map $V = [v_1, \ldots, v_n]$ is also called a **synthesis** map since it puts together (i.e., synthesizes) a function from some numerical information.

Now, if the v_j are elements of some linear space X, and B is any *linear* map defined on X, then

$$B\Big(\sum_{j=1}^n c_j v_j\Big) = \sum_{j=1}^n c_j (Bv_j).$$

In terms of that new notation, this says that

$$\boxed{B[v_1, \ldots, v_n] = [Bv_1, \ldots, Bv_n].}$$

This is exactly what happens when B and V are (compatible) matrices and we compute the product of B with V:

$$(BV)(:,j) = B(V(:,j)),$$

i.e., the jth column of the product is the result of applying B to the jth column of V.

So, to return to our problem of interpolation, we are seeking $\mathbf{c}$ so that $\Lambda(V\mathbf{c}) = \mathbf{b}$, i.e., by the associativity of map composition, so that

$$(1) \qquad (\Lambda V)\mathbf{c} = \mathbf{b}.$$

Here, ΛV is an n-by-n matrix, since it equals $\Lambda[v_1, \ldots, v_n] = [\Lambda v_1, \ldots, \Lambda v_n]$, and each Λv_j is an n-vector. In particular, if the matrix ΛV is invertible, then $\mathbf{c} = (\Lambda V)^{-1}\mathbf{b}$ is the unique solution of (1), i.e., $f := V(\Lambda V)^{-1}\mathbf{b}$ is the unique element in $\operatorname{ran} V$ for which $\Lambda f = \mathbf{b}$. In particular, if $\mathbf{b} = \Lambda g$ for some element g (in the underlying linear space of functions), then

$$f := Pg := V(\Lambda V)^{-1}\Lambda g$$

is the unique element in $\operatorname{ran} V$ that interpolates to g in the sense that $\Lambda f = \Lambda g$.

How did we choose $v_1, \ldots, v_n$? We chose it as a basis for a given linear space. A nontrivial linear space has infinitely many bases to choose from. From the point of view of (1), the ideal choice would make ΛV the identity matrix. If $\Lambda V = \mathrm{id}$, then we say that Λ and V (or, more precisely, the two sequences, $(\lambda_1, \ldots, \lambda_n)$ and $(v_1, \ldots, v_n)$) are **dual to each other**. If we think in terms of V, then we say, in this case, that Λ (or, more precisely, the sequence $(\lambda_1, \ldots, \lambda_n)$) is a **dual basis** for V.

Proposition. *If the data map $\Lambda : f \mapsto (\lambda_i f : i = 1, \ldots, n)$ is dual to the column map $V = [v_1, \ldots, v_n]$, then for any element g (in the underlying linear space of functions),*

$$f := Pg := V\Lambda g$$

is the unique element in $\operatorname{ran} V$ *that interpolates g in the sense that $\Lambda f = \Lambda g$.*

We had several examples of this.

(i) *Lagrange form for polynomial interpolation*: With $\Lambda f := (f(t_1), f(t_2), \ldots, f(t_n))$, and

$$v_j(x) := \prod_{i \neq j} \frac{x - t_i}{t_j - t_i}, \quad j = 1, \ldots, n,$$

we have $v_j(t_i) = \delta_{i,j}$, i.e., $\Lambda V = \mathrm{id}$ (provided $t_i \neq t_j$ for $i \neq j$). Therefore, in that case,

$$V\Lambda g = \sum_{j=1}^{n} g(t_j) v_j$$

is the unique element in $\operatorname{ran} V = \Pi_{n-1}$ that agrees with a given g at $t_1, \ldots, t_n$.

(ii) *Truncated Taylor expansion at some point a*: With

$$\Lambda f := (f(a), f'(a), f''(a), \ldots, f^{(n-1)}(a)),$$

and

$$v_j(x) := (x - a)^{j-1}/(j - 1)!, \quad j = 1, \ldots, n,$$

we have $\lambda_i v_j = \delta_{i,j}$, i.e., $\Lambda V = \mathrm{id}$. Therefore, for given smooth g,

$$f = V\Lambda g = \sum_{j=1}^{n} g^{(j-1)}(a)(\cdot - a)^{j-1}/(j - 1)!$$

is the unique polynomial of degree $< n$ that interpolates g in the sense that $\Lambda f = \Lambda g$, i.e., $f^{(j)}(a) = g^{(j)}(a)$ for $j = 0, \ldots, n - 1$.

(iii) *Least-squares polynomial approximation*: With $v_j = p_{j-1}$ for $j = 1, \ldots, n$, and $p_0, p_1, p_2, \ldots$ a sequence of orthogonal polynomials with respect to some inner product $\langle f, g \rangle$, like $\langle f, g \rangle = \int_a^b f(t)g(t)w(t)\,\mathrm{d}t$ for some positive w, and with

$$\Lambda f := (\frac{\langle f, v_j \rangle}{\langle v_j, v_j \rangle} : j = 1, \ldots, n),$$

we have $\lambda_i v_j = \delta_{i,j}$, i.e., $\Lambda V = \mathrm{id}$. Therefore, for given g,

$$f = V\Lambda g = \sum_{j=1}^{n} v_j \, \frac{\langle g, v_j \rangle}{\langle v_j, v_j \rangle}$$

Is the unique element in $\Pi_{n-1} = \operatorname{ran} V$ that interpolates g in the sense that $\langle f, v_j \rangle = \langle g, v_j \rangle$, $j = 1, \ldots, n$. This says that $f - g$ is perpendicular to $\operatorname{ran} V$, and, since $f \in \operatorname{ran} V$, this says that f is the best approximation to g from $\operatorname{ran} V$ with respect to the norm $\| \cdot \| := \sqrt{\langle \cdot, \cdot \rangle}$.

For a general data map $\Lambda = [\lambda_1; \ldots; \lambda_n]$, it may not be all that easy to choose a basis map V for the given n-dimensional subspace (from which the interpolant is to be chosen) so that $\Lambda V = \mathrm{id}$. But it may be possible to so choose it that the linear system (1) can be 'easily' solved. One such choice is to make

$$A := \Lambda V$$

a diagonally dominant banded matrix. Here, **diagonally dominant** means that $|A(i,i)| > \sum_{j \neq i} |A(i,j)|$ for all i, while **banded** means that, for some k, $A(i,j) = 0$ whenever $|i - j| > k$. The former implies that we need not make any row interchanges when solving the linear system $A\mathbf{c} = \mathbf{b}$ by Gauss elimination, and the latter implies that, in that process, most of the entries we ordinarily have to zero out are already zero, so our work is much reduced. In the extreme case when $k = 0$, A is just a diagonal matrix, hence $c_i = b_i / A(i,i)$, all i. A further additional benefit of a banded A (with 'small' k) is that it takes only $O(n)$ memory to store it. (`MATLAB` provides the special matrix type `sparse` for this.)

An example: a second-order ODE two-point BVP and cubic splines

Here is a nonstandard example of interpolation as used to solve, approximately, an ordinary linear differential equation problem by **collocation**, i.e., by insisting that an approximate solution satisfy the equation exactly at certain (finitely many) points.

The problem is to determine (approximately) the function x that satisfies the *second-order linear differential equation*

$$x''(t) = r(t)x'(t) + s(t)x(t) + z(t), quada \leq t \leq b, \tag{2}$$

together with the *boundary conditions*

$$x(a) = x_a, \quad x'(b) = x'_b,$$

with r, s, z given functions and x_a, x'_b given constants.

In **cubic spline collocation**, one chooses

$$a = \xi_1 < \xi_2 < \cdots < \xi_{\ell+1} = b$$

and, with this, uses the data map

$$\Lambda x := (x(a), ((x'' - rx' - sx)(\xi_i) : i = 1, \ldots, \ell + 1), x'(b)),$$

and seeks a cubic spline x with knot sequence

$$\mathbf{t} = (a, a, a, a, \xi_2, \xi_3, \ldots, \xi_\ell, b, b, b, b)$$

for which

$$\Lambda x = \mathbf{b} := (x_a, (z(\xi_i) : i = 1, \ldots, \ell+1), x'_b), \tag{3}$$

as this guarantees that such an x satisfies the two boundary conditions and also satisfies the differential equation exactly at the points $\xi_1, \ldots, \xi_{\ell+1}$. Indeed, the first equation and last equation in (3) enforces the two boundary conditions, and the remaining $\ell+1$ equations in (3) state that x is to satisfy the differential equation (2) exactly at the $\ell+1$ points ξ_i.

The condition that x is a cubic spline with that specified knot sequence $\mathbf{t}$ is the same as saying that

$$x = [B_1, \ldots, B_n]\mathbf{c}$$

for some sequence $\mathbf{c}$, with $(B_1, \ldots, B_n)$ the sequence of cubic B-splines for that knot sequence. Since $\mathbf{t}$ has $3 + (\ell+1) + 3 = \ell + 7$ entries, this means that

$$n = \#\mathbf{t} - 4 = (\ell+7) - 4 = \ell + 3,$$

hence equal to the number of equations in (3). This means that the linear system

$$A\mathbf{c} := \Lambda[B_1, \ldots, B_n]\mathbf{c} = \mathbf{b}$$

corresponding to (3) has a square coefficient matrix, A. It remains to construct that matrix.

For that, we need to compute ΛB_j for every j and this, in turn, requires the values of B_j, B'_j, and B''_j at all the ξ_i's. Fortunately, `MATLAB`'s spline toolbox provides the command `spcol(`$\mathbf{t}$`,k,`τ`)` that returns the matrix

$$G := \mathrm{M}[B_1, \ldots, B_n]$$

with M the data map

$$\mathrm{M}x := (x^{(m_i)}(\tau_i) : i = 1, 2, \ldots)$$

where

$$m_i := \#\{j < i : \tau_j = \tau_i\}, \quad i = 1, 2, \ldots,$$

and $\tau = (\tau_1 \le \tau_2 \le \tau_3 \le \cdots)$. In particular, with the particular choice

$$\tau = (\xi_1, \xi_1, \xi_1, \xi_2, \xi_2, \xi_2, \ldots, \xi_{\ell+1}, \xi_{\ell+1}, \xi_{\ell+1})$$

(that contains each of the ξ_i exactly 3 times), we get

$$\mathrm{M}x = (x(\xi_1), x'(\xi_1), x''(\xi_1), x(\xi_2), x'(\xi_2), x''(\xi_2), \ldots, x(\xi_{\ell+1}), x'(\xi_{\ell+1}), x''(\xi_{\ell+1})).$$

(An easy way in MATLAB to assemble the sequence τ from the sequence ξ is via the statement

$$\mathtt{tau} = \mathtt{reshape}([\mathtt{xi};\mathtt{xi};\mathtt{xi}], 1, 3*(\mathtt{l}+1));$$

and this assumes that `xi` is the one-row matrix with entries $\xi_1, \ldots, \xi_{l+1}$, and $\mathtt{l} = \ell$.) Note that the resulting matrix

$$G := \mathrm{M}[B_1, \ldots, B_n]$$

has altogether $3(\ell+1)$ rows, with the row-triple $G(3i-2,:)$, $G(3i-1,:)$, and $G(3i,:)$ providing value, first, and second derivative at ξ_i of all the B-splines involved. In particular, the following MATLAB statements will assemble the matrix A from the matrix `G = spcol(`$\mathbf{t}$`,4,`τ`)`:

```
A = zeros(l+3);
A(1,:) = G(1,:);
for i=1:l+1
    A(i+1,:) = G(3*i,:) - r(xi(i))*G(3*i-1,:) - s(xi(i))*G(3*i-2,:):
end
A(l+3,:) = G(end-1,:);
```

Assuming that `b` is the one-column matrix that contains, correspondingly, the vector $\mathbf{b}$ from (3), we get the coefficient vector $\mathbf{c}$ for the spline collocation solution as `c = A\b;`. Hence the statement

`sp = spmak(`$\mathbf{t}$`,(A\b)');`

provides a description of that spline that can be used, as in `vals = fnval(sp,x);` to provide the value(s) of the spline at the point(s) `x`, or as in `fnplt(sp)` to plot the spline, etc. (It is necessary to use in `spmak` the transpose, `(A\b)'`, rather than the one-column matrix `A\b` since `spmak(t,c)` expects the ith coefficient to be `c(:,i)`, i.e., the ith column of the input `c`.)