Interpolation and Approximation

Classic Polynomial Interpolation

Fit a polynomial \( p(x) \) of degree \( n \) (\( p \in \mathbb{P}_n \)) to interpolate a function \( f(x) \) at \( n + 1 \) points \((f(x_0), \ldots, f(x_n))\). That is, find the unique values \( \alpha_0, \ldots, \alpha_n \) such that

\[
\alpha_0 + \alpha_1 x_0 + \ldots + \alpha_n x_0^n = f(x_0) \\
\alpha_0 + \alpha_1 x_1 + \ldots + \alpha_n x_1^n = f(x_1) \\
\vdots \\
\alpha_0 + \alpha_1 x_n + \ldots + \alpha_n x_n^n = f(x_n)
\]

which we write as \( V_n^T[a] = [f] \) where \( V_n \) is a Vandermonde matrix.

The above form can be unstable. To arrive at a better solution, consider the Lagrange form

\[
p(x) = L(x) := \sum_{j=0}^{n} y_j \ell_j(x) \text{ where } \ell_j(x) = \prod_{k \neq j} \frac{x-x_k}{x_j-x_k} \text{ are the Lagrange basis functions,}
\]

and the coefficients \( y_j = f(x_j) \). The problem of finding the Lagrange basis functions and their coefficients can be rewritten in the Barycentric form: Let \( w_j = \frac{1}{\prod_{k \neq j}(x_j-x_k)} \), then

\[
p(x) = \frac{\sum_{j=0}^{n} \frac{w_j}{x-x_j} f(x_j)}{\sum_{j=0}^{n} \frac{w_j}{x-x_j}}.
\]

Using this form, we can multiply a scale factor \( \lambda_k \) against the numerator and denominator throughout the computation process to prevent the \( w_j \) values from over/underflowing (without changing the solution to the problem).

On the other hand, note that we could also pick a single point \( x^* \) and nail down the derivatives of \( f \) at \( x^* \) (i.e., \( f(x^*) = p(x^*), f'(x^*) = p'(x^*), \ldots, f^{(n)}(x^*) = p^{(n)}(x^*) \)) to get the degree \( n \) Taylor approximation to \( f \) at \( x^* \). If we have a mix of derivative information at various points, we can nail down the first \( k \) derivative values at \( n \) points to get a degree \((nk - 1)\) polynomial that interpolates \( f \) and its derivative using a Newton divided difference table of \( f \). This is called Hermite interpolation.

Minimizing Polynomial Interpolation Error

We want to approximate a continuous function \( f \) by a polynomial \( p \), and we want the error to be reasonably small. The good news is that by the Weierstrass Approximation Theorem we can approximate \( f \) to arbitrary precision by some polynomial of some (sufficiently high) degree. The bad news is that by Faber’s Theorem for any interpolation nodes the corresponding polynomial approximation of any degree could be arbitrarily bad.
From the above, it follows that what we really want is the best approximation to \( f(x) \) in the space of all degree \( n \) polynomials. That is, find \( p^* = \min_{p \in \mathbb{P}^n} \| p - f \| \) where \( \| . \| \) is some norm defined on continuous functions.

If we take \( \| . \| \) to be the \( L_{\infty} \) norm, then we arrive at the following theorem:

**Equioscillation Theorem:**

Let \( f \in C[-1, 1] \) (or equivalently in \( C[a, b] \) since we can map \( C[a, b] \) to \( C[-1, 1] \)). The best degree \( n \) polynomial uniform approximation to \( f \) exists, is unique, and is given by \( p^* \in \mathbb{P}^n \) such that the error function \( [f - p^*](x) \) equioscillates at least \( n + 2 \) times.

The approximation error can be written \( |f(x) - p(x)| = K \prod_{k=0}^n (x - x_k) \) where \( K \) is a constant. If we want to minimize the uniform approximation error over all possible \( f \in C[-1, 1] \), we must find nodes \( x_k \) that make the error polynomial \( M(x) = |\prod_{k=0}^n (x - x_k)| \) equioscillate. The solutions to this equioscillation problem are the Chebyshev nodes:

\[
\hat{x}_k = \cos \left( \frac{k\pi}{n} \right) \quad \text{where} \quad k = 0, 1, \ldots, n.
\]

Note that these nodes define a unique polynomial \( T_n \) that has roots at the Chebyshev nodes and whose lead coefficient is 1. The Chebyshev polynomials of differing degrees are all orthogonal, and each is also the uniformly smallest polynomial of degree \( n \) on the interval \([-1, 1]\) with lead coefficient 1. Then the solution \( p^* \) can be written as \( p^*(x) = \sum_{i=0}^n \alpha_i T_i(x) \) where \( T_k(x) \) is the \( k \)th Chebyshev polynomial.

If we take \( \| . \| \) to be any norm, then we have a least squares problem whose solution is the projection of \( f \) onto the space \( \mathbb{P}^n \) (with respect to the corresponding inner product \( \langle ., . \rangle \)). Let \( Q = \{q_0, q_1, \ldots, q_n\} \) be an orthonormal basis for \( \mathbb{P}^n \). Then the projection of \( f \) onto \( \mathbb{P}^n \), \( p^* \), is given by:

\[
p^*(x) = \sum_{i=0}^n \alpha_i q_i \quad \text{where} \quad \langle \sum_{i=0}^n \alpha_i q_i, q_\ell \rangle = \langle f, q_\ell \rangle \quad \text{for all} \quad q_\ell.
\]

The orthonormal basis \( Q \) can be constructed through the Gram-Schmidt orthogonalization process (described in Numerical Analysis Pt. 1). Generally, the inner product \( \langle f, q_\ell \rangle \) will correspond to some integral over a power of \( f \) and \( q_\ell \) multiplied by some weights. Since we don’t know the value of \( f \) other than at discrete points, we have to approximate this integral using a quadrature rule (described later under numerical integration).

**Splines**

Splines are piecewise polynomial approximations. The class of \( k \) times continuously differentiable splines of degree \( n \) is given: \( S^n_k := \{ s \in C^{(k)}[a, b] : s|_{I_k} \in \mathbb{P}^n \} \) where \( I_k \) are disjoint intervals covering some domain \([a, b]\). We want to think of splines in terms of their basis functions such that \( s \in S^n_k \) can be written \( s = \sum_{i=1}^m \alpha_i b_i(x) \) where \( b_i(x) \) is a \( k \)-times differentiable basis function.

Important classes include \( S^0_1 \) (linear splines) with basis “hat” functions \( b_i \) that go linearly from 1 at each node \( x_i \), to 0 at each of \( x_i \)’s neighboring nodes; and \( S^1_3 \) (cubic splines) with
“bubble” basis functions $b_i$ that look like concave quadratic functions with each of their maxima at $(x_i, 1)$ for some interpolation node $x_i$. Since the domain of each basis function ends at each of its neighbors, computing a specific spline in $S^k_m$ amounts to matching the first $k$ derivative with each neighbor and then scaling each basis $b_i$ centered at $x_i$ by the value $f(x_i)$. This can be expressed as solving a tri-diagonal linear system.

**Numerical Integration**

A quadrature rule is a function that approximates the value of a definite integral. In general, the idea is simple: fit a spline function to the underlying function $f$ then approximate $\int_a^b f(x)dx$ by $Q(f) = \sum_k A(I_k)$ where $A(I_k)$ is the area under the segment of the spline-fit defined over the interval $I_k$. If we fit $f$ with splines on uniformly distanced nodes (let the distance between each pair of nodes be a constant $h$), then we get Newton-Cotes quadrature rules. A Newton-Cotes rules where the spline-fits come from the class $S^0_1$ are called trapezoid rules and can be computed easily:

$$Q(f) = \int_a^b f(x)dx \approx \frac{h}{2} [f(x_0) + 2f(x_1) + 2f(x_2) + \ldots + 2f(x_{n-1}) + f(x_n)].$$

If we fit $f$ with a quadratic spline we get Simpson’s rule. For an odd number of nodes, Simpson’s rule yields the following closed form solution:

$$Q(f) = \int_a^b f(x)dx \approx \frac{h}{3} [f(x_0) + 4f(x_1) + 2f(x_2) + \ldots + 4f(x_{n-2}) + 2f(x_{n-1}) + f(x_n)].$$

The degree of exactness of a quadrature rule is the highest order polynomial that the quadrature rule would exactly compute. Every quadrature rule that uses at least $m + 1$ distinct nodes has degree of exactness of at least $m$. The Peano-Kernel error for a quadrature rule $Q$ with degree of exactness $m$ is given by

$$Err(f) := \int_a^b f(x)dx - Q(f) = \int_a^b f^{(m+1)}(t)K(t)dt \quad \text{where} \quad K(t) = \frac{1}{m!}Err_x(x-t)^m.$$ 

Gauss Quadrature rules attempt to minimize error by weighting the value $f(x_i)$ for each $x_i$ so as to minimize the Peano-Kernel error. Clenshaw-Curtis rules choose Chebyshev nodes instead of uniformly distributed nodes in order to minimize interpolation error and therefore also quadrature error.

**Ordinary Differential Equations**

Initial Value Problems: Given $y' = f(t,y)$ and an initial value pair $(t_0, y_0)$, find the original function $y(t)$. Note that since the value of $y'$ is dependent on the current value of $y$ at
each time $t$ (a sort of feedback loop) this is a fundamentally much harder problem than simple quadrature. To approximate a solution, we decompose the problem into a discrete time stepping problem: Given a current iterate $(t_c, y_c)$, find the next point on the trajectory $(t_+, y_+)$. 

To approximate a solution, we could use a quadrature rule:

$$y_+ - y_c = \int_{t_c}^{t_+} f(t, y(t)) dt \approx Q(f).$$

If we use only $f(t_0, y_0)$ and a single step (two node linear) Quadrature rule, then we get Euler’s method:

$$y_+ \approx y_c + hf(t_c, y_c) \quad \text{where } h = t_+ - t_c.$$ 

More generally, we can compute the weighted average of any number of “slopes” $f(t^*, y(t^*))$ where $t_c \leq t^* \leq t_+$. Let the weights be denoted $b_k$, and let the various corresponding step sizes (as measured from $t_c$) be denoted $c_k$. Also, note that for $\ell \geq 2$ approximate slopes, we can use a weighted average of the previous $\ell - 1$ slopes to approximate the $\ell$th slope of $y(t)$. We will call the $k \leq \ell - 1$ weights associated with weighting the previous slopes in the $\ell$th slope approximation $a_{\ell k}$. Then:

$$f(t_{\ell}, y_{\ell}) \approx \sum_{k=1}^{\ell-1} a_{\ell k} f(t_c + c_k, \xi_k)$$

where $\xi_k$ is the $k$th slope approximation for $f$. Then the final step $y_+$ is given by the weighted average:

$$y_+ = y_c + h \sum_{i=1}^{s} b_i \xi_i$$ 

where $h$ is the full step size. Since we are using $s$ approximate slopes total, this is called an $s$-stage Runge-Kutta method.

Note, we can fully describe every Runge-Kutta method by a table called its Butcher Tableau. The following is a Butcher Tableau for a 4-stage Runge-Kutta method:

<table>
<thead>
<tr>
<th>$c_1$</th>
<th>$b_1$</th>
<th>$0$</th>
<th>$0$</th>
<th>$0$</th>
<th>$0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_2$</td>
<td>$a_{21}$</td>
<td>$0$</td>
<td>$0$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>$c_3$</td>
<td>$a_{31}$</td>
<td>$a_{32}$</td>
<td>$0$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
<tr>
<td>$c_4$</td>
<td>$a_{41}$</td>
<td>$a_{42}$</td>
<td>$a_{43}$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

Note that all the $a_{kj}$ on or above the diagonal must be zero since we can’t use slopes that haven’t been approximated yet. Also, the $c_i$ must be in the range $[0, h]$ since $h$ is the complete step size, typically with $c_0 = 0$ and $c_f = h$ (where $c_f$ is the final step size, in this case $f = 4$). A necessary and sufficient condition for the Runge-Kutta method to be consistent (i.e., it converges to the exact solution as $h \to 0$) is for $\sum_{i=1}^{n} b_i = 1$ (the weights must sum to one).
If it is also true that each \( c_\ell = \sum_{k=1}^{\ell-1} a_{\ell k} \), then we say that the method is invariant. This means that it can be \textit{autonomized}. That is, by doing a change of variables and setting:

\[
z' = \begin{bmatrix} y' \\ T \end{bmatrix} = \begin{bmatrix} f(T(t), y(t)) \\ 1 \end{bmatrix},
\]

The differential equation can be reduced to an autonomous differential equation that doesn’t depend on a time variable. This is a favourable property.

**Root-Finding (Optimization)**

Note that Optimization is equivalent to root finding on the derivative. The following are efficient root finding algorithms for Lipschitz continuous functions.

A contraction is a mapping \( G : D \to D \) satisfying \( G(D) \subseteq D \). For every contraction, a fixed point \( \xi \) exists such that \( G(\xi) = \xi \) and it is unique. Given a contraction \( G : D \to D \), find a fixed-point in \( \xi \in D \) such that \( G(\xi) = \xi \). If \( G \) is Lipschitz continuous, then the sequence \( x_{k+1} = G(x_k) \) will converge to \( \xi \) linearly (at least) at the rate of the Lipschitz constant. One way to find a root of \( F \), define \( G(x) = x - F(x) \). Then \( G(x) \) is a contraction and its fixed point \( G(\xi) = \xi \) corresponds to a root of \( F \).

Another way (for locally linear functions) is to use Newton’s method. Let \( F \) be the function, then the truncation of Taylor’s series to its linear approximation lets us jump directly to an approximate solution. If \( F \) is monotone then this defines a contraction:

\[
x_{k+1} = x_k - \nabla F(x_k)^{-1} F(x_k).
\]

If \( F \)’s derivative is locally Lipschitz, then this contraction will converge locally quadratically. More typically, use an approximation \( M_k \approx \nabla F(x_k)^{-1} \), then use the (fast) Broyden update:

\[
M_{k+1} = M_k + \frac{(\Delta F_k - M_k \Delta x_k) \Delta x_k^T}{\|\Delta x_k\|_2^2}.
\]