NE 155, Classes 7 & 8, S16
Interpolation and Approximation
February 3 and 5, 2016

Introduction

Sometimes rather than calculating values of a function, we need to approximate or interpolate them instead. Why might we do this?

- It may be difficult or impossible to analytically evaluate the function
- We may have only a table of values at certain points and need to determine the values in between
- It may be much faster to compute values of an approximate function than of the original function (important if we have to calculate the function values over and over again)
- A function may be defined implicitly

Interpolation and/or approximation is used in numerical **integration/differentiation** and in **solving differential equations**.

Problem: given a function $f(x)$ or a set of discrete values $\{x_i, f(x_i)\}$, try to find $g(x)$ that in some sense represents $f(x)$.

1. **Approximation** An approximation function does not need to fit through all given function values.

   Why would we do this? Perhaps we know there are errors in the data, or we only care about the general trend.

   Example: Least Squares

2. **Interpolation** An interpolation function does need to go through all given values of the function.

   We do this because we *believe* the data points and therefore want to make sure we match them. We will discuss this strategy first.

Types

There are several common types of Interpolating functions

- **Polynomial**: $g(x) = P(x) = \sum_i a_i x^i$
- **Piecewise polynomial**: slap some polynomials together so their ends meet with some degree of smoothness
- **Trig functions**: e.g., Fourier series $g(x) = \frac{1}{2}a_0 + \sum_k [a_k \sin(kx) + b_k \cos(kx)]$
- Combination of **rational functions**: $g(x) = R_n(x)/R_m(x)$
- **Exponential functions**: $g(x) = \sum_i a_i \exp(b_i x^i)$

Polynomial Interpolation

Why polynomials?

$$g(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

1. derivatives and integrals are easy to compute $\rightarrow$ after either operation we still have a polynomial
2. efficient to evaluate
$$g(x) = a_0 + x(a_1 + x(a_2 + x(\dots)))$$

Theorem: (Weierstrauss approximation theorem)

let f be $C^0 \in [a, b]$. Given $\epsilon > 0$, $\exists$ a polynomial, $P(x)$, such that

$$|f(x) - P(x)| \leq \epsilon, \qquad x \in [a, b] \, .$$

Aside about terminology: In mathematical analysis, *smoothness* has to do with how many derivatives of a function exist and are continuous. The function f is said to be of (differentiability) class C^k if the derivatives $f', f'', \cdots, f^{(k)}$ exist and are continuous (the continuity is implied by differentiability for all the derivatives except for $f^{(k)}$).

Taylor Polynomials

If $f(x)$ is known but difficult to evaluate, we can replace it by a Taylor polynomial around $x = a$

$$\begin{aligned} f(x) &= f(a) + f'(a)(x-a) + f''(a)\frac{(x-a)^2}{2!} + \cdots + f^{(n-1)}(a)\frac{(x-a)^{n-1}}{(n-1)!} + R_n \\ &= \sum_{k=0}^{n-1} \frac{(x-a)^k}{k!} f^{(k)}(a) + R_n \\ R_n &= f^{(n)}(x^*)\frac{(x-a)^n}{n!} \qquad \text{for some } x^* \in [a, x] \end{aligned}$$

[finite difference]

Lagrange Interpolating Polynomials

Theorem: Instead, if we have $(n+1)$ distinct points $x_0, x_1, \ldots, x_n$ and $f(x)$ is a function defined by these points, then $\exists$ a unique polynomial, $P_n(x)$, of degree $\leq n$ that interpolates f at the $n+1$ distinct points s.t.

$$f(x_k) = P(x_k)\,, \qquad \text{for } k = 0, 1, \ldots, n$$

This polynomial is given by

$$P(x) = f(x_0)L_0(x) + \cdots + f(x_n)L_n(x) = \sum_{k=0}^{n} f(x_k)L_k(x)\,,$$

where the $L_k(x)$ are Lagrange polynomials:

$$\begin{aligned} L_k(x) &= \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{(x - x_i)}{(x_k - x_i)} \\ L_k(x) &= \frac{(x-x_0)(x-x_1)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_n)}{(x_k-x_0)(x_k-x_1)\cdots(x_k-x_{k-1})(x_k-x_{k+1})\cdots(x_k-x_n)} \end{aligned}$$

Note: this is sometimes called full degree polynomial interpolation.

iPython demo; show Taylor picture as well:
http://en.wikipedia.org/wiki/Taylor_series#mediaviewer/File:Sintay_SVG.svg.

Error

Theorem: If $x_0, x_1, \ldots, x_n$ are $(n+1)$ distinct points $\in [a,b]$ and f is $C^{n+1} \in [a,b]$ then for each $x \in [a,b]$ there exists $\xi(x) \in [a,b]$ s.t.

$$f(x) - P_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-x_0)(x-x_1)\cdots(x-x_n)$$

$$\boxed{e = \frac{f^{(n+1)}(\xi)}{(n+1)!}\prod_{i=0}^{n}(x-x_i)}$$

What is a variable here that will affect the error?
How we choose the x_i.
We will not get into this here, but in general equally-spaced points don't work well and there are specific sets of differently-spaced points that are designed to be optimal in different ways.

Piecewise Polynomials

It is often difficult to get one polynomial to fit all of our data very well, and we'd rather not have to select optimal points (since it will depend on the norm we use and this can be tricky).

An alternative approach is to divide the interval into small sub-intervals (usually of equal size) and construct a polynomial on each-subinterval. This is called "splines".

A **spline** is a polynomial function that is piecewise-defined, and possesses a "high" degree of smoothness at the places where the polynomial pieces connect (which are known as *knots*).

The simplest way to do this is **piecewise linear**.
What might be a problem with that?
Piecewise linear is only C^0.

The most common approach is to use **cubic splines**: a cubic polynomial between each successive pair of nodes. For cubic splines, we can get continuous 2nd derivatives at the knots - which is all we can ask from cubic polynomials.

Spline interpolation: Given a function $f(x)$ defined on $[\alpha, \beta]$ and a set of numbers (knots) $\alpha = x_0 < x_1 < ... < x_n = \beta$, we can say the following about a cubic spline interpolant, $S(x)$, for f:

- on each sub-interval $[x_j, x_{j+1}]$, S is a cubic polynomial:

$$S(x) = S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3 \text{ for } j = 0, 1, ..., n - 1$$

- S interpolates f (gives n+1 constraints)

$$S(x_j) = f(x_j) \text{ for } j = 0, 1, ..., n$$

- S is continuous (gives n-1 constraints)

$$S_{j+1}(x_{j+1}) = S_j(x_{j+1}) \text{ for } j = 0, 1, ..., n - 2$$

- S' is continuous (gives n-1 constraints)

$$S'_{j+1}(x_{j+1}) = S'_j(x_{j+1}) \text{ for } j = 0, 1, ..., n - 2$$

- S'' is continuous (gives n-1 constraints)

$$S''_{j+1}(x_{j+1}) = S''_j(x_{j+1}) \text{ for } j = 0, 1, ..., n - 2$$

With n subintervals we have $4n$ unknowns and only $4n - 2$ constraints!

Thus, we select one of these boundary conditions:

- "not-a-knot": remove two analysis points
- "free" or "natural": $S''(x_0) = S''(x_n) = 0$
- "clamped" or "complete": $S'(x_0) = f'(x_0)$ and $S'(x_n) = f'(x_n)$

Which of these do you expect to be the most accurate and why? The least?

- "not-a-knot": least accurate because you're removing data
- "free" or "natural": introduces $O(h^2)$ error near the end points.
- "clamped" or "complete": most accurate, but we need to know the end point derivatives.

$$e = \max_{\alpha \le x \le \beta} |f(x) - S(x)| \le \frac{5}{384} h^4 M$$

where

$$M = \max_{\alpha \leq x \leq \beta} |f(x)^{(4)}|$$

and

$$h = \max_{0 \leq j \leq n-1} (x_{j+1} - x_j)$$

We're not going to go over the spline algorithm, but you get a **tridiagonal system** out of it—which is nice because these are fairly easy to solve directly.

The Python function for this is *scipy.interpolate.splrep*

Approximation

What if we have some collection of data, $(x_0, y_0), (x_1, y_1), \ldots, (x_m, y_m)$ that is noisy but obeys, what we believe, is some functional form (linear, exp., trig., etc.)?

We can approximate the data to get an expression for the functional form.

Least Squares

Least Squares finds the best approximating line (function) that minimizes the least squares error of the representation.

Linear

If we assume the form is linear, we are trying to find

$$y = a_0 + a_1 x \ ,$$

where these coefficients provide the best fit for

$$
\begin{aligned}
y_0 &= a_0 + a_1 x_0 \\
y_1 &= a_0 + a_1 x_1 \\
&\vdots \\
y_m &= a_0 + a_1 x_m
\end{aligned}
$$

$$
\begin{pmatrix} 1 & x_0 \\ 1 & x_1 \\ \vdots & \vdots \\ 1 & x_m \end{pmatrix} \underbrace{\begin{pmatrix} a_0 \\ a_1 \end{pmatrix}}_{\vec{z}} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{pmatrix}
$$

Why can't we just solve this matrix system?
$n < m$ and the system is overdetermined.
Instead, we look to minimize the error, $e = ||\mathbf{A}\vec{z} - \vec{b}||$.

We must choose a norm in which to minimize the error. It is quite common to choose the 2-norm; the 2-norm squared is

$$
e = \sum_{i=0}^{m} \big(y_i - (a_o + a_1 x_i)\big)^2
$$

To minimize this w.r.t. the as, we take the partial derivative and set equal to zero:

$$
\begin{aligned}
\frac{\partial e}{\partial a_0} &= -2 \sum_{i=0}^{m} \big(y_i - (a_0 + a_1 x_i)\big) = 0 \\
\frac{\partial e}{\partial a_1} &= -2 \sum_{i=0}^{m} x_i \big(y_i - (a_0 + a_1 x_i)\big) = 0
\end{aligned}
$$

This gives

$$
\begin{pmatrix} m & \sum_{i=0}^{m} x_i \\ \sum_{i=0}^{m} x_i & \sum_{i=0}^{m} x_i^2 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} \sum_{i=0}^{m} y_i \\ \sum_{i=0}^{m} x_i y_i \end{pmatrix}
$$

and now we have two equations with two unknowns!

note: If you look at the $\mathbf{A}$ we had originally defined, $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}^T\vec{b}$ also gives this system. These

are called the **normal equations**:

$$\begin{aligned}
\mathbf{A}^T\mathbf{A}\vec{z} &= \mathbf{A}^T\vec{b} \\
\det(\mathbf{A}^T\mathbf{A}) &= m\sum_{i=0}^{m} x_i^2 - \left(\sum_{i=0}^{m} x_i\right)^2 \\
&= \frac{1}{2}\sum_{i=0}^{m}\sum_{j=0}^{m}(x_i - x_j)^2
\end{aligned}$$

The normal equation is that which minimizes the sum of the square differences between the left and right sides. It is called a normal equation because $\vec{b} - \mathbf{A}\vec{x}$ is normal to the range of $\mathbf{A}$.

Note that all of this was for a linear function. We can use the same strategy for other functional forms.

Exponential

If the functional form is

$$\begin{aligned}
y &= \beta e^{\alpha x} \qquad \text{or} \\
y &= \beta x^{\alpha}
\end{aligned}$$

You can do the least square procedure by taking the two-norm squared, taking the partial derivatives, and setting them equal to zero to solve

-or-

you can simply take the log and treat it like the linear case

$$\begin{aligned}
\underbrace{ln(y)}_{y} &= \underbrace{ln(\beta)}_{a_0} + \underbrace{\alpha}_{a_1} x \qquad \text{or} \\
\underbrace{ln(y)}_{y} &= \underbrace{ln(\beta)}_{a_0} + \underbrace{\alpha}_{a_1}\underbrace{ln(x)}_{x}
\end{aligned}$$

Polynomial

We can also find the least squares error for some polynomial approximation:

$$y = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

where the error (residual) in the two norm is given by

$$e = \sum_{i=0}^{m} \left(y_i - (a_o + a_1 x_i + \cdots + a_n x_i^n) \right)^2 .$$

We again take the partial derivatives and form the matrices for the normal equations (or directly form the normal equations using matrix multiplication). This gives:

$$\begin{pmatrix} m & \sum_{i=0}^m x_i & \cdots & \sum_{i=0}^m x_i^n \\ \sum_{i=0}^m x_i & \sum_{i=0}^m x_i^2 & \cdots & \sum_{i=0}^m x_i^{n+1} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=0}^m x_i^n & \sum_{i=0}^m x_i^{n+1} & \cdots & \sum_{i=0}^m x_i^{2n} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} \sum_{i=0}^m y_i \\ \sum_{i=0}^m x_i y_i \\ \vdots \\ \sum_{i=0}^m x_i^n y_i \end{pmatrix}$$

Fourier Series

Least Squares and Fourier Series are actually related to one another.

It is possible to approximate a function with a polynomial, as we know, but now we will write the way we're considering the error a bit differently:

$$e = \int_a^b \left[f(x) - P_n(x) \right]^2 dx$$

$$\text{where } P_n(x) = \sum_{j=0}^{n} c_j \phi_j(x)$$

In general, approximations to functions require that the basis set (the ϕs) be orthogonal.

A **basis** is a linearly independent set of vectors. Given a basis of a vector space, every element of the vector space can be expressed uniquely as a finite linear combination of basis vectors.

$\{\phi_0, \phi_1, \ldots, \phi_n\}$ is an orthogonal set of functions over $[a, b]$ w.r.t. a weight function, $w(x)$, if

$$\int_a^b w(x)\phi_j(x)\phi_k(x)dx = \begin{cases} 0 & \text{if } j \neq k \\ \alpha_x & \text{if } j = k \end{cases} .$$

If $\alpha = 1$, then the functions are also orthonormal.

If there is an orthogonal set of functions on $[a, b]$, then the least squares approximation to f on $[a, b]$ is:

$$\begin{aligned} P(x) &= \sum_{j=0}^{n} c_j(x)\phi_j(x) \\ c_j &= \frac{1}{\alpha_j}\int_a^b w(x)\phi_j(x)f(x)dx \qquad k = 0, 1, \ldots, n . \end{aligned}$$

To see the relationship with Fourier series, we use the orthonormal basis set on $[-\pi, \pi]$:

$$\begin{aligned} \phi_0(x) &= \frac{1}{\sqrt{2\pi}} \\ \phi_j(x) &= \frac{1}{\sqrt{\pi}}\cos(jx) , \qquad j = 1, 2, \ldots, n \\ \phi_{n+j}(x) &= \frac{1}{\sqrt{\pi}}\sin(jx) , \qquad j = 1, 2, \ldots, n-1 . \end{aligned}$$

Then, the least squares approximation to f on $[-\pi, \pi]$ with this set of functions is

$$\begin{aligned} S_n(x) &= \sum_j = 0^{2n-1} c_j(x)\phi_j(x) \\ c_j &= \int_{-\pi}^{\pi} \phi_j(x)f(x)dx , \qquad k = 0, 1, \ldots, 2n-1 , \end{aligned}$$

and in the limit as $n \to \infty$, $S_n(x) \to$ the Fourier series of $f(x)$.