MATH3511/6111: Scientific Computing

03. Interpolation

Lindon Roberts

Semester 1, 2022

Office: Hanna Neumann Building #145, Room 4.87
Email: lindon.roberts@anu.edu.au
Based on lecture notes written by S. Roberts, L. Stals, Q. Jin, M. Hegland, K. Duru.

Australian National University

Approximation of Functions

- Functions are complicated mathematical objects
  - Can be any mapping from inputs to outputs, not just those with a simple form based on standard mathematical operations
- Many real-world quantities are described by functions
  - Temperature in Canberra (vs. time), stock prices (vs. time), ocean depth (vs. location), etc.
- In practice, usually don't have functions represented by equations, but by data points

Two important tasks:

- Approximate a given function with a simpler function — easier to calculate things we care about (e.g. derivative, integral, max/min).
- Interpolation: find a function which fits some given data. (regression: fit to data with errors)

Types of Function Approximations

When we approximate/interpolate a function, we usually want it to be of a simple type. Common types of functions used include:

- Polynomials: $f(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$
- Trignometric functions (e.g. Fourier series): $f(x) = a_0 + a_1 \cos x + a_2 \cos(2x) + \cdots$
- Exponentials: $f(x) = a_1 e^{-b_1(x-x_1)^2} + \cdots + a_n e^{-b_n(x-x_n)^2}$
- Piecewise polynomials

We will focus on polynomials in this course.

Theorem (Weierstrass Approximation Theorem)

If $f : [a, b] \to \mathbb{R}$ is a continuous function, then for any tolerance $\epsilon > 0$, there exists a polynomial p such that $|f(x) - p(x)| < \epsilon$ for all $x \in [a, b]$.

The degree of $p(x)$ may be extremely high, which is not practical. We will focus on low(ish) degree polynomials.

Evaluating Polynomials

Simple Question

If we have a polynomial

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

how do we evaluate $p(x)$ for a given value of x?

This is easy:

```
def evaluate_polynomial(a, x, n):
    # a is a list/array of coefficients, length n+1
    p = a[0]
    for k in range(1, n+1):  # loop from k=1 to k=n (inclusive)
        p = p + a[k] * x**k
    return p
```

Computing x^k takes $k-1$ multiplications, so overall this procedures takes n additions and $1+2+\cdots+n = n(n+1)/2$ multiplications.

Evaluating Polynomials

If we store x^k as a variable, we can be more efficient:

```
def evaluate_polynomial2(a, x, n):
    # a is a list/array of coefficients, length n+1
    p = a[0]
    x_pow_k = 1.0
    for k in range(1, n+1):  # loop from k=1 to k=n (inclusive)
        x_pow_k = x_pow_k * x  # equal to x**k
        p = p + a[k] * x_pow_k
    return p
```

This now requires n additions and $2n$ multiplications.

It turns out that we can do better!

Horner's Method

An efficient way of evaluating polynomials is given by Horner's method (William Horner, 1819 but known much earlier, back to China c. 200AD).

Write $p(x)$ as

$$p(x) = a_0 + x\left(a_1 + x\left(a_2 + x\left(\cdots + x(a_{n-1} + a_n x)\right)\right)\right).$$

```
def horner(a, x, n):
    # a is a list/array of coefficients, length n+1
    p = a[n]
    for k in range(n):  # loop from k=0 to k=n-1 (inclusive)
        p = a[n-1-k] + x * p  # k=0 gives a[n-1], ..., k=n-1 gives a[0]
    return p
```

This only needs n multiplications and n additions (about 30% faster than our previous method).

Interpolation

Our basic problem is interpolation (also called collocation).

Interpolation Problem

Given points $(x_0, y_0), \ldots, (x_n, y_n)$, find a function $f(x)$ of a particular type such that

$$f(x_i) = y_i, \qquad \text{for all } i = 0, \ldots, n.$$

The points x_i are called the nodes, and the values y_i are called the ordinates.

Polynomial Interpolation

We want to do interpolation with polynomials.

Theorem

If all $x_0, \ldots, x_n$ are distinct, then for any values of $y_0, \ldots, y_n$ there exists a unique polynomial $p_n(x)$ of degree at most n which satisfies the interpolation conditions

$$p_n(x_i) = y_i, \qquad \text{for all } i = 0, \ldots, n.$$

This theorem is easiest to see by trying to explicitly construct $p_n(x)$.

Polynomial Interpolation

Suppose we have

$$p_n(x) = a_0 + a_1 x + \cdots + a_n x^n,$$

and we impose the conditions $p_n(x_i) = y_i$ for $i = 0, \ldots, n$. We get the following linear system:

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

This $(n+1) \times (n+1)$ matrix is called a Vandermonde matrix (Alexandre-Théophile Vandermonde, early 1770s).

This matrix A is invertible when all x_i are distinct: if $A\boldsymbol{z} = \boldsymbol{0}$ then $p(x) = z_0 + z_1 x + \cdots + z_n x^n$ is a polynomial of degree n with $n+1$ distinct roots. Hence $p(x) = 0$ and so $\boldsymbol{z} = \boldsymbol{0}$.

Polynomial Interpolation: Example

Suppose we want to find a quadratic $p_2(x) = a_0 + a_1 x + a_2 x^2$ interpolating the data

i	0	1	2
x_i	0	0.5	2
y_i	0.2	0.6	-1

All x_i are distinct, so this gives the invertible linear system

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0.5 & 0.25 \\ 1 & 2 & 4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 0.2 \\ 0.6 \\ -1 \end{bmatrix}.$$

The unique solution is: $a_0 = 1/5$, $a_1 = 19/15$ and $a_2 = -14/15$.

Therefore the (unique) interpolating quadratic is

$$p_2(x) = \frac{1}{5} + \frac{19}{15}x - \frac{14}{15}x^2.$$

Representation of Polynomials

Recall: the set of polynomials of degree at most n forms a vector space of dimension $n+1$.

One basis for this vector space is the set of monomials: $1, x, x^2, \ldots, x^n$. Writing

$$p(x) = a_0 + a_1 x + \cdots + a_n x^n,$$

is just writing $p(x)$ in terms of this basis.

However, for interpolation it is often useful to use a different basis.

Lagrange Polynomials

A clever choice of basis for interpolation is the set of Lagrange polynomials (Joseph-Louis Lagrange, 1795, but originally Edward Waring, 1779).

For nodes $x_0, \ldots, x_n$, the j-th Lagrange polynomial is the degree n polynomial which is zero at all nodes except one:

$$L_j(x_i) = \begin{cases} 1 & i = j, \\ 0 & i \neq j. \end{cases}$$

There are $n + 1$ Lagrange polynomials, one corresponding to each node.

Calculating Lagrange Polynomials

How do we find the Lagrange polynomials for $x_0, \ldots, x_n$?

We know that $L_j(x)$ is degree n and has n roots, since $L_j(x_i) = 0$ for all $i = 0, \ldots, n$ except for $i = j$. Therefore,

$$L_j(x) = c_j(x - x_0)(x - x_1)\cdots(x - x_{j-1})(x - x_{j+1})\cdots(x - x_n).$$

We then find c_j by using the last interpolation condition, $L_j(x_j) = 1$, which gives

$$c_j = \frac{1}{(x_j - x_0)(x_j - x_1)\cdots(x_j - x_{j-1})(x_j - x_{j+1})\cdots(x_j - x_n)}.$$

This means that

$$L_j(x) = \frac{(x - x_0)(x - x_1)\cdots(x - x_{j-1})(x - x_{j+1})\cdots(x - x_n)}{(x_j - x_0)(x_j - x_1)\cdots(x_j - x_{j-1})(x_j - x_{j+1})\cdots(x_j - x_n)} = \prod_{\substack{i=0 \\ i\neq j}}^{n} \frac{x - x_i}{x_j - x_i}.$$

Calculating Lagrange Polynomials

Example

Calculate the Lagrange polynomials for nodes $x_0 = 0$, $x_1 = 0.5$ and $x_2 = 2$.

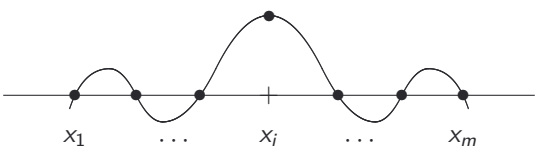
$$L_0(x) = \frac{(x-0.5)(x-2)}{(0-0.5)(0-2)} = (x-0.5)(x-2),$$
$$L_1(x) = \frac{(x-0)(x-2)}{(0.5-0)(0.5-2)} = -\frac{4x(x-2)}{3},$$
$$L_2(x) = \frac{(x-0)(x-0.5)}{(2-0)(2-0.5)} = \frac{x(x-0.5)}{3}.$$

Lagrange Interpolation

Interpolating data $(x_0, y_0), \ldots, (x_n, y_n)$ is easy once we have calculated the Lagrange polynomials:

$$p_n(x) = y_0 L_0(x) + y_1 L_1(x) + \cdots + y_n L_n(x) = \sum_{i=0}^{n} y_i L_i(x).$$

This is known as the Lagrange form of the interpolating polynomial.



Lagrange Form: Example

Example

Find the Lagrange form of the quadratic $p_2(x)$ interpolating the data

i	0	1	2
x_i	0	0.5	2
y_i	0.2	0.6	-1

The nodes are $x_0 = 0$, $x_1 = 0.5$ and $x_2 = 2$. We calculated the corresponding Lagrange polynomials on the previous slide. Therefore the interpolating polynomial is

$$p_2(x) = 0.2L_0(x) + 0.6L_1(x) - L_2(x).$$

This is the same polynomial as we found earlier (using Vandermonde matrix), but written in a different basis. If we were given new data with the same x_i but different y_i, it is easier to change the Lagrange form.

Newton Form

Another way to represent the polynomial is the Newton form. For this, we build a sequence of polynomials $p_0(x), \ldots, p_n(x)$ such that $p_k(x)$ has degree $k+1$ and interpolates the data $(x_0, y_0), \ldots, (x_k, y_k)$.

Firstly, we get $p_0(x)$ is a zero-th order polynomial (i.e. constant function) with $p_0(x_0) = y_0$. This gives $p_0(x) = y_0$.

Then, by induction, suppose we have $p_{k-1}(x)$ already. We choose $p_k(x)$ to have the form

$$p_k(x) = p_{k-1}(x) + c_k(x - x_0)(x - x_1)\cdots(x - x_{k-1}).$$

Clearly $p_k(x_j) = p_{k-1}(x_j)$ for $j = 0, \ldots, k-1$, so p_k interpolates $(x_0, y_0), \ldots, (x_{k-1}, y_{k-1})$ by induction.

We only need to find c_k such that $p_k(x_k) = y_k$.

Newton Form: Example

Example

Find the Newton form of the quadratic $p_2(x)$ interpolating the data

i	0	1	2
x_i	0	0.5	2
y_i	0.2	0.6	-1

First, we have $p_0(x) = y_0 = 0.2$. Then,

$$p_1(x) = p_0(x) + c_1(x - 0) = 0.2 + c_1 x.$$

Substituting $p_1(0.5) = 0.6$ we get $c_1 = 0.8$, so $p_1(x) = 0.2 + 0.8x$. Lastly,

$$p_2(x) = p_1(x) + c_2(x - 0)(x - 0.5) = 0.2 + 0.8x + c_2 x(x - 0.5).$$

Substituting $p_2(2) = -1$ we get $c_2 = -14/15$, and so

$$p_2(x) = 0.2 + 0.8x - \frac{14}{15}x(x - 0.5).$$

Again, this is the same quadratic as before, but written in a different polynomial basis.

Lagrange vs. Newton

When to use Lagrange or Newton forms?

- Use Lagrange if you expect to interpolate new y_i with the same x_i.
- Use Newton if you expect to receive extra data $(x_{n+1}, y_{n+1}), \ldots$ (just need to perform more inductive steps)

In practice, interpolating using Vandermonde is not a good idea for large n (matrix becomes very ill-conditioned — discussed later in the course).

Lagrange vs. Newton

With some work (which we skip), we can rewrite the Lagrange form as

$$\boxed{p_n(x) = \frac{\sum_{i=0}^{n} \frac{w_i}{x-x_i} y_i}{\sum_{i=0}^{n} \frac{w_i}{x-x_i}}}$$

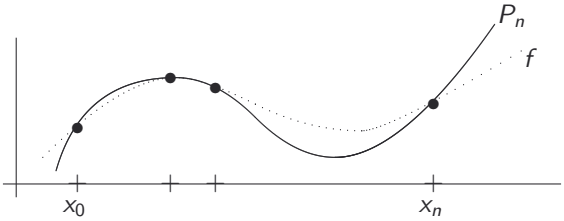
This is known as the barycentric formula for $p_n(x)$, where we have defined

$$w_i = \frac{1}{(x_i - x_0)(x_i - x_1)\cdots(x_i - x_{i-1})(x_i - x_{i+1})\cdots(x_i - x_n)}, \qquad \text{for } i = 0, \ldots, n.$$

The barycentric formula has both benefits: it is easy to change the data y_i, or to add new data (x_{n+1}, y_{n+1}). Also, if the nodes are equally spaced (or based on Chebyshev points, below), there are simple formulae for the w_i.

This is implemented in Python in `scipy.interpolate.BarycentricInterpolator`.

Polynomial Interpolation Error Analysis



The error formula for polynomial interpolants is similar to the error formula for Taylor series.

Theorem

If $f(x)$ is $n+1$ times continuously differentiable, then for any x_0 and x there is some ξ between x and x_0 such that

$$f(x) = f(x_0) + f'(x_0)(x-x_0) + \frac{f''(x_0)}{2}(x-x_0)^2 + \cdots + \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n + \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-x_0)^{n+1}.$$

Polynomial Interpolation Error Analysis

For polynomial interpolation, the corresponding theorem is:

Theorem

Suppose $f(x)$ is $n+1$ times continuously differentiable and all $x_0, \ldots, x_n$ are distinct. Then if $y_i = f(x_i)$, the (unique) polynomial interpolant $p_n(x)$ of degree at most n satsifies

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

for some ξ (depending on x) in the interval containing all x_i and x.

This bound is mostly useful if we know something about the size of the higher-order derivatives of $f(x)$.

Polynomial Interpolation: Example

Let's try to interpolate $f(x) = \sin x$ using n equally spaced points between -3 and 3:

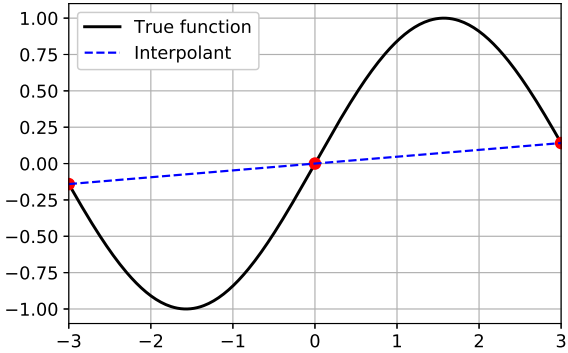


Figure 1. $n = 3$ points (quadratic interpolant).

Polynomial Interpolation: Example

Let's try to interpolate $f(x) = \sin x$ using n equally spaced points between -3 and 3:

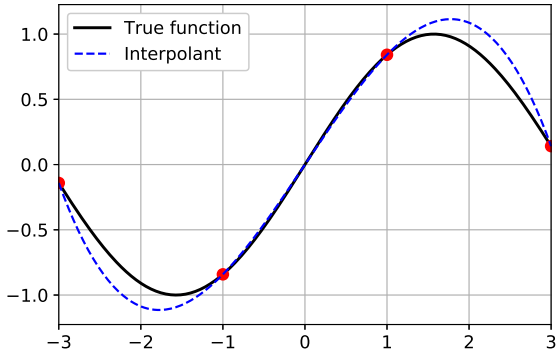


Figure 2. $n = 4$ points (cubic interpolant).

Polynomial Interpolation: Example

Let's try to interpolate $f(x) = \sin x$ using n equally spaced points between -3 and 3:

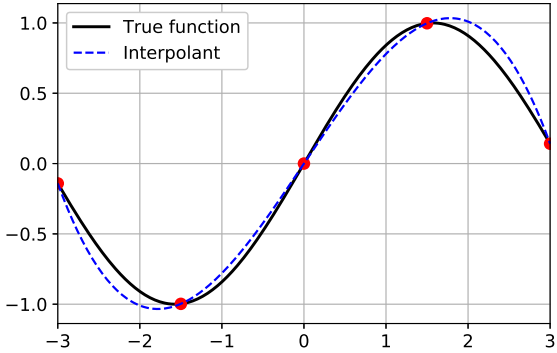


Figure 3. $n = 5$ points.

Polynomial Interpolation: Example

Let's try to interpolate $f(x) = \sin x$ using n equally spaced points between -3 and 3:

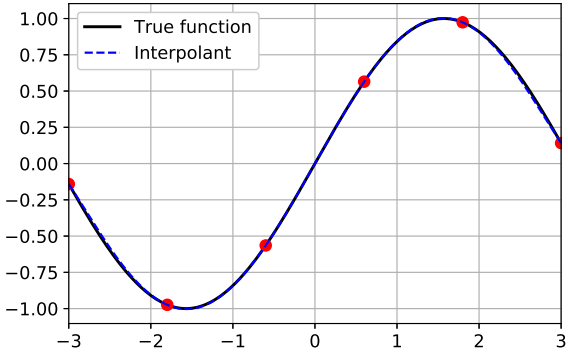


Figure 4. $n = 6$ points.

Polynomial Interpolation: Example

Let's try to interpolate $f(x) = \sin x$ using n equally spaced points between -3 and 3:

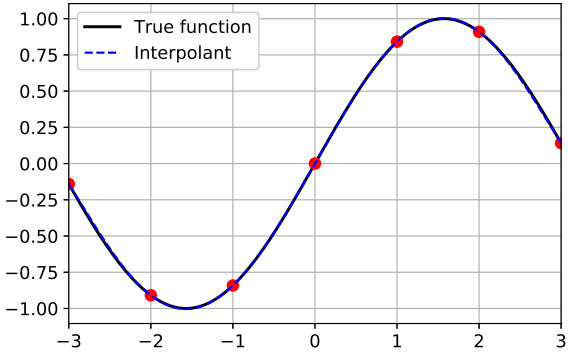


Figure 5. $n = 7$ points.

Polynomial Interpolation: Example

A table of the maximum error $|f(x) - p_n(x)|$ for $x \in [-3, 3]$, depending on n:

n	Max error
3	0.9272163215301206
4	0.2105964314462818
5	0.1498471254862008
6	0.0199043025434354
7	0.0140934327085960
8	0.0011429657930386
9	0.0008137842776297
10	0.0000442264476727

Higher degree polynomials approximate $f(x)$ better, as we might expect.

Polynomial Interpolation: Another Example

Now, let's interpolate $f(x) = 1/(1 + 25x^2)$ using n equally spaced points between -1 and 1:

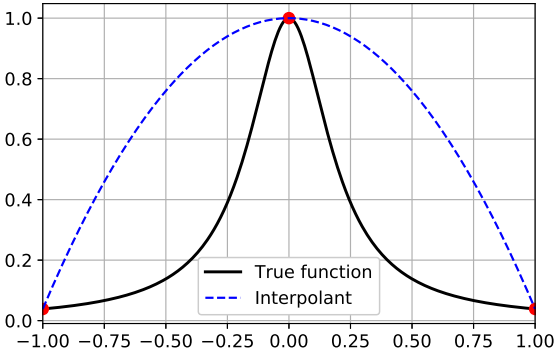


Figure 6. $n = 3$ points (quadratic interpolant).

Polynomial Interpolation: Another Example

Now, let's interpolate $f(x) = 1/(1 + 25x^2)$ using n equally spaced points between -1 and 1:

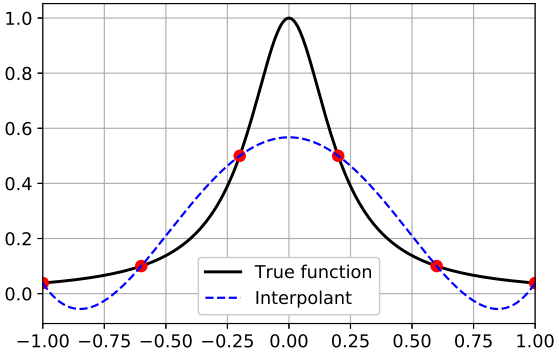


Figure 7. $n = 6$ points.

Polynomial Interpolation: Another Example

Now, let's interpolate $f(x) = 1/(1 + 25x^2)$ using n equally spaced points between -1 and 1:

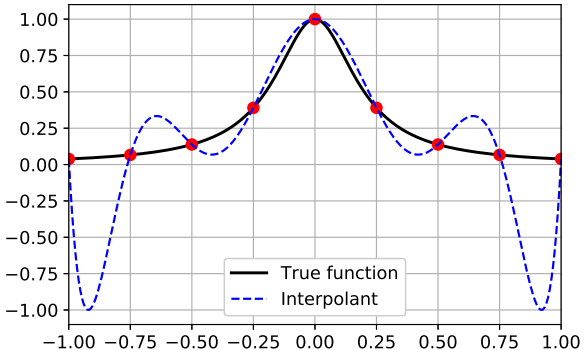


Figure 8. $n = 9$ points.

Polynomial Interpolation: Another Example

Now, let's interpolate $f(x) = 1/(1 + 25x^2)$ using n equally spaced points between -1 and 1:

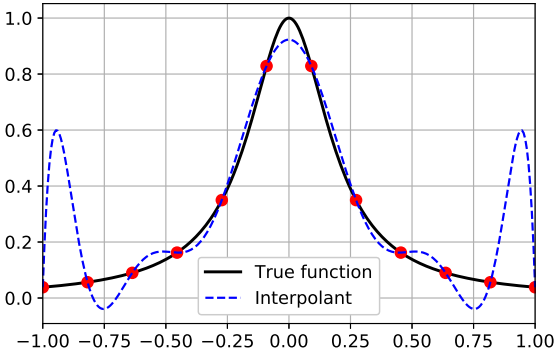


Figure 9. $n = 12$ points.

Polynomial Interpolation: Another Example

Now, let's interpolate $f(x) = 1/(1 + 25x^2)$ using n equally spaced points between -1 and 1:

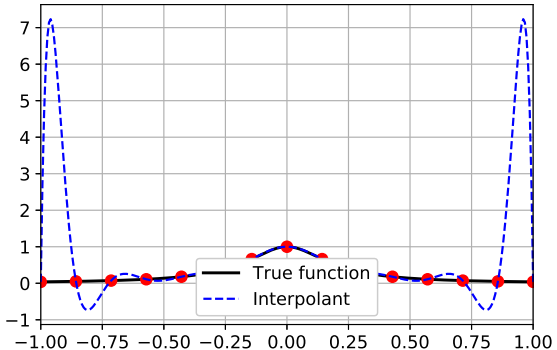


Figure 10. $n = 15$ points.

Polynomial Interpolation: Example

A table of the maximum error $|f(x) - p_n(x)|$ for $x \in [-1, 1]$, depending on n:

```
 n        Max error
 3    0.6462290923465023
 4    0.7067368609091735
 5    0.4383215021335619
 6    0.4324321064060296
 7    0.6166202190184811
 8    0.2471306833699505
 9    1.0451573170837032
10    0.3002677757562047
11    1.9156006511727122
12    0.5563570799959867
13    3.6631019905132942
14    1.0682823428804211
15    7.1912080076604292
...
30  328.0340350908162463
```

Higher degree polynomials approximate $f(x)$ worse! What's going on?

Polynomial Interpolation: Another Example

The error in the interpolation is:

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

Let's plot the derivatives of $f(x)$:

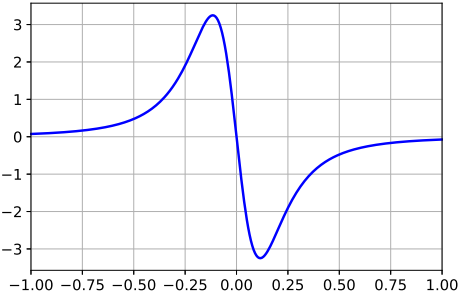


Figure 11. Plot of 1st derivative of Runge function $f(x) = 1/(1 + 25x^2)$.

Polynomial Interpolation: Another Example

The error in the interpolation is:

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

Let's plot the derivatives of $f(x)$:

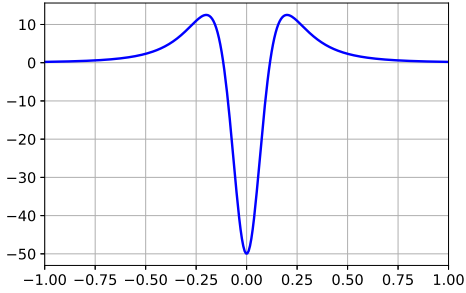


Figure 12. Plot of 2nd derivative of Runge function $f(x) = 1/(1 + 25x^2)$.

Polynomial Interpolation: Another Example

The error in the interpolation is:

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

Let's plot the derivatives of $f(x)$:

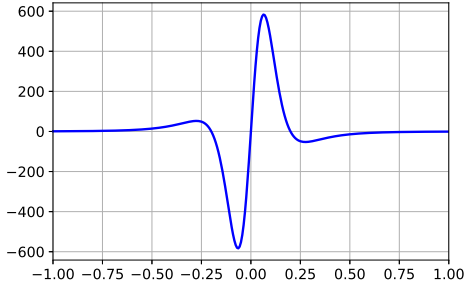


Figure 13. Plot of 3rd derivative of Runge function $f(x) = 1/(1 + 25x^2)$.

Polynomial Interpolation: Another Example

The error in the interpolation is:

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

The higher-order derivatives of $f(x)$ get very large! This means our error might get large whenever $\prod_{i=0}^{n}(x - x_i)$ is also large:

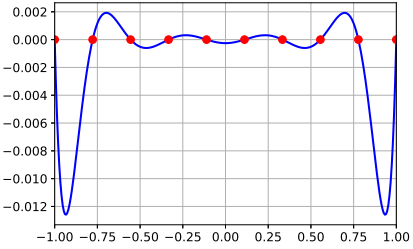


Figure 14. Plot of $\prod_{i=0}^{n}(x - x_i)$ for 10 equally spaced points.

Chebyshev Points

There are two reasons that polynomial interpolation didn't work for this function for x near ± 1:

- Its higher-order derivatives are large (in magnitude)
- The term $\prod_{i=0}^{n}(x - x_i)$ is large(r) near $x = \pm 1$.

There is nothing we can do about the first problem, but the second can be addressed by choosing different points (not equally spaced).

The optimal placement of the points is so that they are mostly clustered near ± 1:

$$x_i = \cos\left(\frac{2i+1}{2n+2}\pi\right), \qquad \text{for } i = 0, \ldots, n.$$

Or, if we want to approximate a function on another interval $[a, b]$,

$$x_i = \frac{a+b}{2} + \frac{b-a}{2}\cos\left(\frac{2i+1}{2n+2}\pi\right), \qquad \text{for } i = 0, \ldots, n.$$

These are called the Chebyshev points (Pafnuty Chebyshev, 1854).

Chebyshev Points

For the interval $[-1, 1]$, the Chebyshev points are:

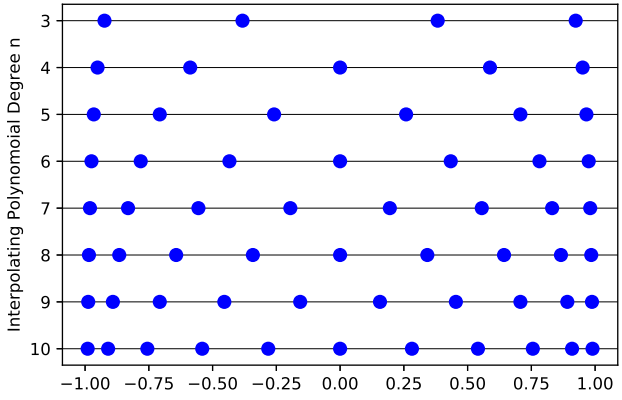


Figure 15. Locations of Chebyshev points on $[-1, 1]$ for different polynomial degrees n.

Chebyshev Points

The Chebyshev points make $\prod_{i=0}^{n}(x - x_i)$ small everywhere:

$$\left|\prod_{i=0}^{n}(x - x_i)\right| \leq \frac{1}{2^n}, \qquad \text{for all } x \in [-1, 1].$$

This is the best possible bound for any choice of interpolation points.

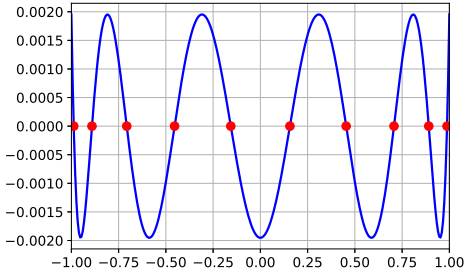


Figure 16. Plot of $\prod_{i=0}^{n}(x - x_i)$ for 10 Chebyshev points.

Chebyshev Points

Now, let's interpolate $f(x) = 1/(1 + 25x^2)$ using n Chebyshev points between -1 and 1:

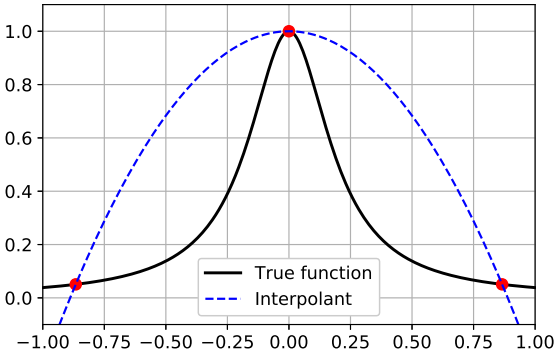


Figure 17. $n = 3$ Chebyshev points (quadratic interpolant).

Chebyshev Points

Now, let's interpolate $f(x) = 1/(1 + 25x^2)$ using n Chebyshev points between -1 and 1:

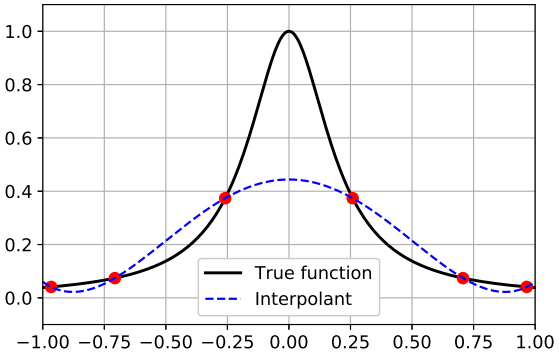


Figure 18. $n = 6$ Chebyshev points.

Chebyshev Points

Now, let's interpolate $f(x) = 1/(1 + 25x^2)$ using n Chebyshev points between -1 and 1:

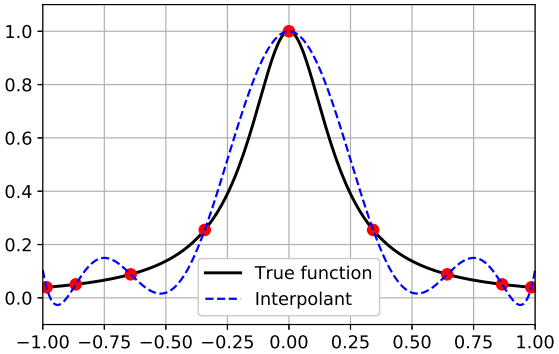


Figure 19. $n = 9$ Chebyshev points.

Chebyshev Points

Now, let's interpolate $f(x) = 1/(1 + 25x^2)$ using n Chebyshev points between -1 and 1:

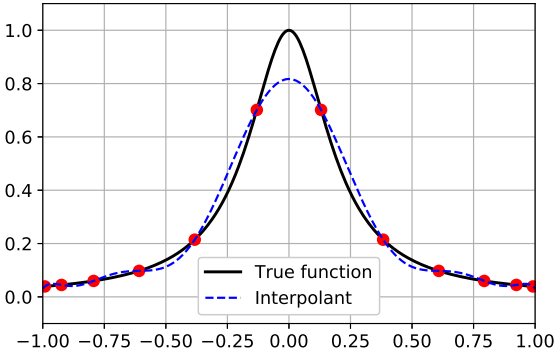


Figure 20. $n = 12$ Chebyshev points.

Chebyshev Points

Now, let's interpolate $f(x) = 1/(1 + 25x^2)$ using n Chebyshev points between -1 and 1:

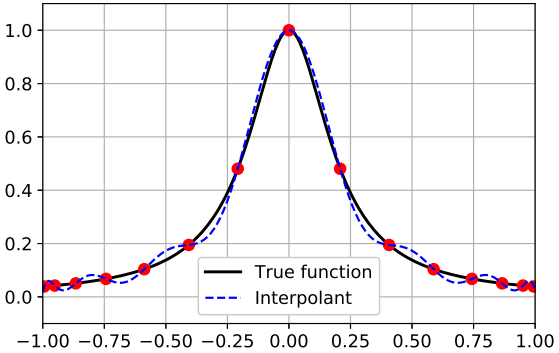


Figure 21. $n = 15$ Chebyshev points.

Chebyshev Points

A table of the maximum error $|f(x) - p_n(x)|$ for $x \in [-1, 1]$, depending on n:

```
 n        Max error
 3    0.6005977354703360
 4    0.7502994320440761
 5    0.4020169231313263
 6    0.5559106745862767
 7    0.2642272913179964
 8    0.3917396641284535
 9    0.1708355459656833
10    0.2691777743703794
11    0.1091535004653675
12    0.1827577893778712
13    0.0692154832732549
14    0.1233972502213379
15    0.0466022882026120
...
30    0.0051560939355677
```

Using Chebyshev points fixes the problem! Now the interpolation error decreases as n increases.

Chebyshev Points

If we use Chebyshev points for interpolation with the barycentric formula (above), we get an explicit expression for the w_i:

$$w_i = (-1)^i \sin\left(\frac{2i+1}{2n+2}\pi\right), \qquad \text{for } i = 0, \ldots, n.$$

This makes it very easy to construct the barycentric interpolant using Chebyshev points, so is a good 'default choice' for polynomial interpolation.

By comparison, for equally spaced points, we get the formula

$$w_i = (-1)^i \binom{n}{i}, \qquad \text{for } i = 0, \ldots, n,$$

which can be very large (which leads to the "Runge phenomenon" of large errors near the endpoints).

Chebyshev Polynomials

Where do the Chebyshev points come from? We wanted to make $\prod_{i=0}^{n}(x - x_i)$ small for all $x \in [-1, 1]$.

The Chebyshev polynomials are the polynomials which take the smallest maximum values on $[-1, 1]$. They are given by

$$T_n(x) = \cos(n \arccos(x))$$

Using various trigonometric identities, we get

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x),$$

and so

$$T_0(x) = 1, \quad T_1(x) = x, \quad T_2(x) = 2x^2 - 1, \quad T_3(x) = 4x^3 - 3x, \quad T_4(x) = 8x^4 - 8x^2 + 1, \ldots$$

So to make $\prod_{i=0}^{n}(x - x_i)$ small, we take x_i to be the roots of the Chebyshev polynomial of degree $n + 1$. This gives the optimal bound $\left|\prod_{i=0}^{n}(x - x_i)\right| \leq 2^{-n}$ for all $x \in [-1, 1]$.

Chebyshev Polynomials

The function $T_n(x)$ has n roots in $[-1, 1]$, at

$$x = \cos\left(\frac{k - \frac{1}{2}}{n}\pi\right), \qquad \text{for } k = 1, \ldots, n,$$

and $n+1$ extrema (maxima where $T_n(x) = 1$ & minima where $T_n(x) = -1$) at

$$x = \cos\left(\frac{k}{n}\pi\right), \qquad \text{for } k = 0, 1, \ldots, n.$$

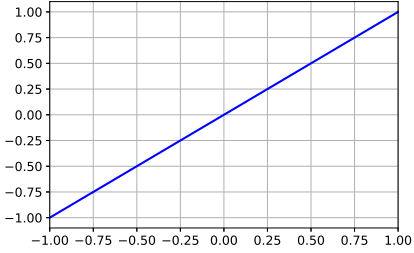


Figure 22. Plot of $T_1(x)$

Chebyshev Polynomials

The function $T_n(x)$ has n roots in $[-1, 1]$, at

$$x = \cos\left(\frac{k - \frac{1}{2}}{n}\pi\right), \qquad \text{for } k = 1, \ldots, n,$$

and $n + 1$ extrema (maxima where $T_n(x) = 1$ & minima where $T_n(x) = -1$) at

$$x = \cos\left(\frac{k}{n}\pi\right), \qquad \text{for } k = 0, 1, \ldots, n.$$

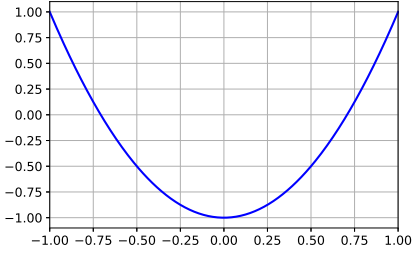


Figure 23. Plot of $T_2(x)$

Chebyshev Polynomials

The function $T_n(x)$ has n roots in $[-1, 1]$, at

$$x = \cos\left(\frac{k - \frac{1}{2}}{n}\pi\right), \qquad \text{for } k = 1, \ldots, n,$$

and $n + 1$ extrema (maxima where $T_n(x) = 1$ & minima where $T_n(x) = -1$) at

$$x = \cos\left(\frac{k}{n}\pi\right), \qquad \text{for } k = 0, 1, \ldots, n.$$

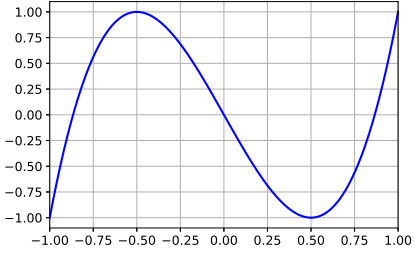


Figure 24. Plot of $T_3(x)$

Chebyshev Polynomials

The function $T_n(x)$ has n roots in $[-1, 1]$, at

$$x = \cos\left(\frac{k - \frac{1}{2}}{n}\pi\right), \qquad \text{for } k = 1, \ldots, n,$$

and $n + 1$ extrema (maxima where $T_n(x) = 1$ & minima where $T_n(x) = -1$) at

$$x = \cos\left(\frac{k}{n}\pi\right), \qquad \text{for } k = 0, 1, \ldots, n.$$

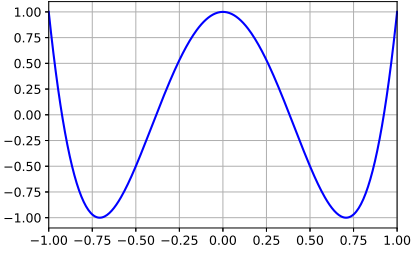


Figure 25. Plot of $T_4(x)$

Chebyshev Polynomials

The function $T_n(x)$ has n roots in $[-1, 1]$, at

$$x = \cos\left(\frac{k - \frac{1}{2}}{n}\pi\right), \qquad \text{for } k = 1, \ldots, n,$$

and $n + 1$ extrema (maxima where $T_n(x) = 1$ & minima where $T_n(x) = -1$) at

$$x = \cos\left(\frac{k}{n}\pi\right), \qquad \text{for } k = 0, 1, \ldots, n.$$

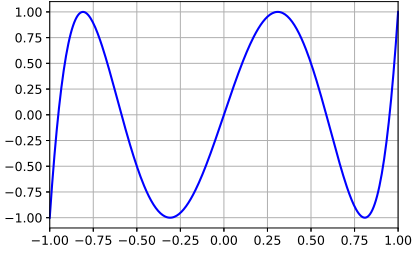


Figure 26. Plot of $T_5(x)$

Chebyshev Polynomials

The function $T_n(x)$ has n roots in $[-1, 1]$, at

$$x = \cos\left(\frac{k - \frac{1}{2}}{n}\pi\right), \qquad \text{for } k = 1, \ldots, n,$$

and $n+1$ extrema (maxima where $T_n(x) = 1$ & minima where $T_n(x) = -1$) at

$$x = \cos\left(\frac{k}{n}\pi\right), \qquad \text{for } k = 0, 1, \ldots, n.$$

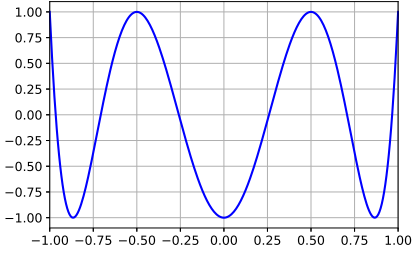


Figure 27. Plot of $T_6(x)$