1 The Interpolation Problem

In this section, we will discuss two related problems that underlie mathematical techniques for data fitting and modeling: function interpolation and approximation. In this set of notes, we will discuss the first: what does it mean to interpolate a function, what families of functions do we use for this and what methods exist for it.

For the purposes of this entire section, we will be making the following assumptions on our data:

Definition 1 *We will assume we have $\{x_0, x_1, \ldots, x_n\} \in \mathbb{R}$ locations, and we will typically assume they are distinct. We measure a dependent variable $y = f(x)$ at these locations, yielding the* interpolation data*:*

$$y_j = f(x_j) \quad j = 0, 1, \ldots, n$$

We will assume that $f(x)$ is a smooth function (with as many continuous derivatives as we need) and crucially, that the measured data are accurate evaluations of this function; that is, it is not noisy*, and the only errors come from floating point representation / arithmetic.*

Given these assumptions, it makes sense to try to find some function $p(x)$ such that it goes through each one of our data points. We can then use this $p(x)$ to perform calculations we would need the unknown $f(x)$ for: evaluation between the points, derivatives, integrals, etc.

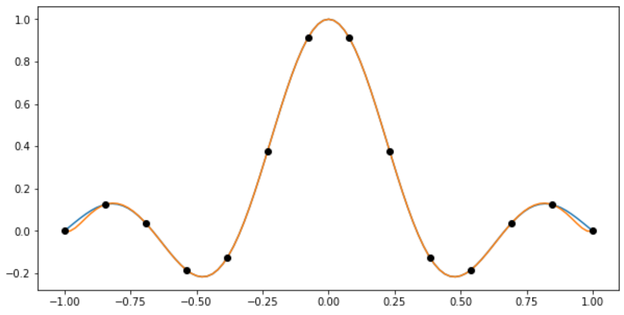

Figure 1.1: Polynomial interpolant of the sinc function for $n + 1 = 13$ nodes

Definition 2 *Given the interpolation data in Definition 1, a function $p(x)$ is said to interpolate the data, or to be an interpolant of $f(x)$, if*

$$p(x_j) = f(x_j) = y_j \quad j = 0, 1, \ldots, n$$

Given $n+1$ function evaluations, there are clearly many candidate functions $p(x)$ that would go through the data and that could, depending on our application, be useful: polynomials, piecewise polynomials, trigonometric functions, exponentials, splines, etc. We must then specify a family of functions for which the interpolation problem has a unique solution (and hopefully that solution is stable to small changes in the data).

2 Polynomial Interpolation

We know from our algebra and calculus classes that, given 2 points on the plane, a unique line (polynomial of degree ≤ 1) goes through them. Given 3 points, a unique quadratic (polynomial of degree ≤ 2) does the job. We also observe that the number of coefficients in these polynomials matches the number of data points.

From this, we may hypothesize that given $n+1$ points as in Definition 1, there is a unique polynomial of degree $\leq n$ that interpolates this data. We define this space of polynomials and establish our notation

Definition 3 *We denote $\mathcal{P}_n = \{p(x) \text{ polynomial} \mid \deg(p) \leq n\}$ the vector space of polynomials of degree $\leq n$. A canonical basis for this vector space is $\{1, x, x^2, \ldots, x^n\}$, and so given $p \in \mathcal{P}_n$, there is a unique $a \in \mathbb{R}^n$ such that*

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

If we write down a polynomial in $\mathcal{P}_n$ in the canonical basis, and write down what it means for that polynomial to interpolate the data, we get a system of $n+1$ linear equations in the unknown coefficients a_i:

$$\begin{aligned} a_0 + a_1 x_0 + a_2 x_0^2 + \cdots + a_n x_0^n &= y_0 \\ a_0 + a_1 x_1 + a_2 x_1^2 + \cdots + a_n x_1^n &= y_1 \\ &\vdots \\ a_0 + a_1 x_n + a_2 x_n^2 + \cdots + a_n x_n^n &= y_n \end{aligned}$$

In matrix form, this becomes:

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

$$\mathcal{V}\mathbf{a} = \mathbf{y}$$

Where $\mathcal{V}$ is known as the Vandermonde matrix. We note that the columns of the Vandermonde matrix are each canonical basis term evaluated at the interpolation points, $\mathcal{V}(i,j) = x_{i-1}^{j-1}$. We can show that the determinant of the Vandermonde matrix is:

$$\det(\mathcal{V}) = \prod_{j>i} (x_j - x_i)$$

This is non-zero if and only if the interpolation locations are distinct. If this is the case, given any data vector $\mathbf{y} \in \mathbb{R}^{n+1}$, there is a unique set of coefficients $\mathbf{a}$ such that the corresponding polynomial interpolates the data.

We could end here by suggesting a method: form the Vandermonde matrix, and solve the system (e.g. using Gaussian elimination). This is $O(n^3)$, but more importantly... for many location distributions, and certainly for equispaced points, the Vandermonde matrix is horribly conditioned. For equispaced points, $\kappa(\mathcal{V})$ can be shown to grow exponentially fast: by $n = 30$, $\kappa(\mathcal{V}) \geq 10^{16}$ and we get no digits back (that we can trust).

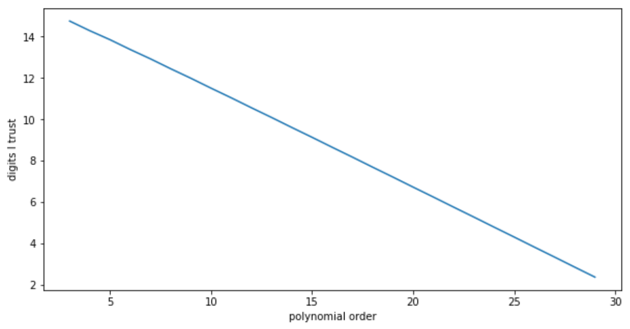

Figure 2.1: Plot of $-log_{10}\varepsilon_M - \log_{10}\kappa(\mathcal{V})$, that is, how many digits we can trust when solving for the polynomial coefficients using Vandermonde

2.1 Lagrange interpolation

Since Vandermonde is not a good idea both in terms of computation and in terms of stability, we want to find a different way to determine the polynomial interpolant. One general idea is: if the canonical basis is "not good for interpolation", maybe a different basis designed for this purpose will do this job better.

Given a set of $n+1$ interpolation data locations x_j, we define the *Lagrange polynomial basis* $L_j \in \mathcal{P}_n$ as the unique polynomials that satisfy:

$$L_j(x_i) = \delta_{i,j} \quad j = 0, 1, \dots, n \tag{2.1}$$

Where $\delta_{i,j}$ is the Kronecker delta, equal to 0 when $i \neq j$ and equal to 1 when $i = j$. Let's say we have found these polynomials.

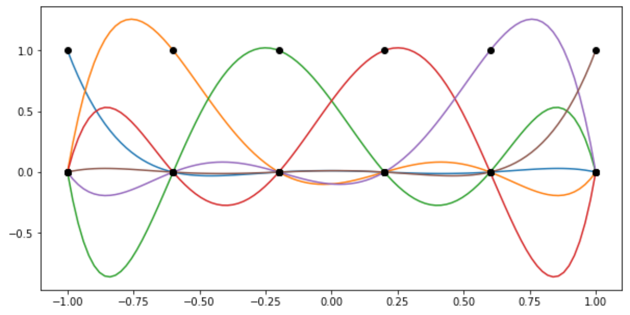

Figure 2.2: Lagrange basis for $n = 5$ equispaced nodes

The idea is then to write the interpolant $p(x)$ as a linear combination of the Lagrange basis terms. Plugging in the interpolation condition, we find:

$$p(x) = \sum_{j=0}^{n} a_j L_j(x)$$
$$y_i = p(x_i) = \sum_{j=0}^{n} a_j L_j(x_i) = a_i$$
$$p(x) = \sum_{j=0}^{n} y_j L_j(x)$$

That is, the function values y_j are the coefficients! It may seem that we have traded one interpolation problem for our data for $n+1$ interpolation problems (to form the Lagrange basis). However, the reason this is not a problem is because we can come up with a general formula for $L_j(x)$.

To see how this works, let's start with the case $n = 1$ (two points). $L_0(x)$ is a line that goes through the points $(x_0, 1)$ and $(x_1, 0)$, and $L_1(x)$ is another one that goes through $(x_0, 0)$ and $(x_1, 1)$. We can easily come up with formulas by setting the roots first, and then multiplying by a constant to make the polynomial evaluate to 1 at the x_i point:

$$L_0(x) = \frac{x - x_1}{x_0 - x_1}$$
$$L_1(x) = \frac{x - x_0}{x_1 - x_0}$$

We can extend this idea to the general case. We find that:

$$L_j(x) = \frac{\prod_{i \neq j}(x - x_i)}{\prod_{i \neq j}(x_j - x_i)} \tag{2.2}$$

That is: the numerator is a product of n linear factors needed to set the roots of $L_j(x)$: x_i for $i \neq j$, and the denominator is a constant set so that $L_j(x_j) = 1$.

Let's say we want to evaluate the interpolant $p(x)$ at a set of m target locations $z_1, \ldots, z_m$; usually these are different from the interpolation locations. Most of the computational work required has to do with evaluating $L_j(z_k)$ accurately for $j = 0, \ldots, n$ and $k = 1, \ldots, m$; this takes $O(mn^2)$ work. We can store these on a $n + 1 \times m$ matrix $\mathbf{L}$. We can then obtain the vector of evaluations by computing the matrix-vector product $p(\mathbf{z}) = \mathbf{Ly}$ ($O(mn)$ work).

2.2 Newton interpolation

The Lagrange basis gives us a way to write down a formula for the polynomial interpolant: we don't have to solve for the coefficients, as they are just our data y_j. In other words: all of the work goes to evaluating the Lagrange polynomials (or other quantities we may be interested in, e.g. derivatives).

There is another important basis for interpolation: the Newton polynomial basis. This basis is simpler and can be evaluated efficiently, but it requires some work to find the coefficients (the Vandermonde matrix for this basis is lower triangular, not the identity).

Given interpolation points, the Newton basis is defined as:

$$\begin{aligned}
\nu_0(x) &= 1\\
\nu_1(x) &= (x - x_0)\\
\nu_2(x) &= (x - x_0)(x - x_1)\\
&\vdots\\
\nu_n(x) &= \prod_{j=1}^{n-1}(x - x_j)
\end{aligned}$$

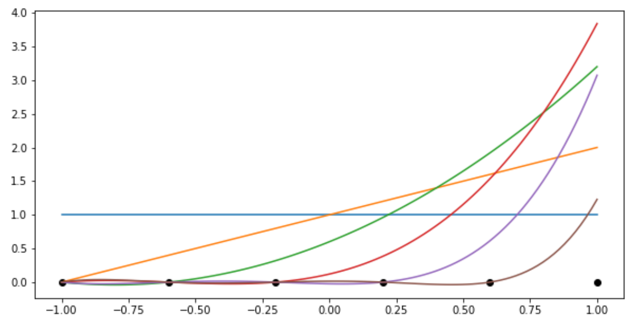

Figure 2.3: Newton basis for $n = 5$ equispaced nodes

We can immediately note that the Newton basis is such that the j-th basis term only depends on the first j locations: $x_0, \ldots, x_{j-1}$. This is going to be true of the interpolation coefficients as well. As such, the Newton basis naturally lends itself to starting with some number of points, and then "updating" our interpolant by adding more locations and function observations.

We note a couple of things: If we form the Vandermonde matrix for Newton, it is lower triangular:

$$\mathcal{V}_{Newton} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 1 & (x_1 - x_0) & 0 & \cdots & 0 \\ 1 & (x_2 - x_0) & (x_2 - x_0)(x_2 - x_1) & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & (x_n - x_0) & (x_n - x_0)(x_n - x_1) & \cdots & \prod_{j=1}^{n-1}(x_n - x_j) \end{bmatrix} \tag{2.3}$$

We notice by multiplying diagonal entries that $\det(\mathcal{V}_{Newton}) = \prod_{i>j}(x_i - x_j)$, same as that for the original Vandermonde matrix. It can be shown that the linear transformation to change the polynomial basis from the canonical basis to the Newton basis is upper triangular, with diagonal elements all equal to 1 (exercise to the reader: find this matrix!). This means the Newton basis can be interpreted as an LU factorization applied to the original Vandermonde matrix, and it means this provides a proof for the determinant of both matrices.

Second: Let the interpolant in $\mathcal{P}_n$ be written as:

$$p(x) = \sum_{j=0}^{n} d_j \nu_j(x) \tag{2.4}$$

because this matrix is triangular, this means we can find the coefficients by a forward triangular solve. This starts as follows:

$$\begin{aligned} f(x_0) &= y_0 = d_0 \\ f(x_1) &= y_1 = f(x_0) + d_1(x_1 - x_0) \\ \frac{f(x_1) - f(x_0)}{x_1 - x_0} &= d_1 \end{aligned}$$

We notice the first coefficient is a function evaluation at x_0, and the second one is a difference of values of $f(x)$ at our first two points, one that would converge to $f'(x_0)$ if x_1 approached x_0. Furthermore, the polynomial $d_0\nu_0(x)+d_1\nu_1(x) = f(x_0)+\frac{f(x_1)-f(x_0)}{x_1-x_0}(x-x_0)$ is a linear approximation of our function that uses the slope of the secant line (instead of the tangent line).

We introduce some notation, and then we state both a formula and an algorithmic way to find the Newton coefficients:

Definition 4 *Given points $x_0, x_1, \ldots, x_n$, we define $f[x_0, \ldots, x_j]$ the Newton divided difference of order $j+1$ recursively as:*

$$\begin{aligned} f[x_0] &= f(x_0) \\ f[x_0, x_1] &= \frac{f[x_1] - f[x_0]}{x_1 - x_0} \\ &\vdots \\ f[x_0, x_1, \ldots, x_j] &= \frac{f[x_1, x_2, \ldots, x_j] - f[x_0, x_2, \ldots, x_{j-1}]}{x_j - x_0} \end{aligned}$$

That is, as a difference of differences of order j.

These differences are precisely the coefficients of the Newton interpolant, which may be found following the forward triangular solve we started above (Exercise: show the formula is correct for the third coefficient / divided difference of order 2). This also gives us an algorithmic way to sequentially compute these coefficients: The Newton tableau (table):

$$\begin{bmatrix} f(x_0) & f[x_0,x_1] & f[x_0,x_1,x_2] & \cdots & f[x_0,x_1,\cdots,x_n] \\ f(x_1) & f[x_1,x_2] & f[x_1,x_2,x_3] & \iddots & \\ \vdots & \vdots & \iddots & & \\ f(x_{n-2}) & f[x_{n-2},x_{n-1}] & f[x_{n-2},x_{n-1},x_n] & & \\ f(x_{n-1}) & f[x_{n-1},x_n] & & & \\ f(x_n) & & & & \end{bmatrix} \tag{2.5}$$

This is a triangular table, where each column is computed as differences of the previous column. We have to be careful in computing it to divide by the corresponding difference of values of x_j. The coefficients for our interpolant written in terms of the Newton basis can be found at the top row, and if we want to add a new point (and thus need a new coefficient), we just have to add a new entry on the anti-diagonal of this table (one entry on each column going from the lower left to the top right corners).

2.2.1 The Horner evaluation algorithm

We note that evaluation of the Newton interpolant can be made very efficient using a variation of the **Horner method** for polynomial evaluation (which also works for the canonical basis, and it is more stable than the expanded versions like the one we saw in our homework). Assuming we have computed all our coefficients, the Horner rule factorizes the polynomial as:

$$p(x) = d_0 + (x - x_0)\left(d_1 + (x - x_1)\left(d_2 + \cdots \left(d_{n-1} + (x - x_{n-1})d_n\right)\cdots\right)\right) \tag{2.6}$$

The algorithm then precedes to evaluate from the inside of these nested parenthesis to the outside:

```
Given $z$ targets :
$p_z = d_n$
for $j = (n-1) : -1 : 0$
    $p_z = d_j + (z - x_j)p_z$
end
```

Bonus: The ordering of Newton points matters for stability

When we code our first Newton divided differences routine and test it for sequentially ordered equispaced nodes (or even for Chebyshev nodes, which as we will see below, are better for interpolation), we notice that this computation is not very stable: for degrees around 30-40, we see numerical error drift towards the right endpoint of our interval (our error curves "look wrong"). This is because we are taking differences of values of f which are close to each other, and then dividing by small quantities (difference between the x_i's).

This issue can be studied more closely by studying the condition number of this divided differences calculation, which can be related to Gaussian elimination performed on the Newton-Vandermonde matrix (which is lower triangular). We can then ask the question: how does the stability of this procedure depend on the order of our interpolation nodes?

In Higham's "Accuracy and Stability of Numerical Methods", this question is explored in some detail. We get the following useful results:

1. For equispaced nodes or nodes which are clustered towards the center, an *inverse central ordering* is optimal. For example, if $x_i = linspace(a, b, n+1)$, we re-order from the highest to the lowest distance to the midpoint $(a + b)/2$.

2. For Chebyshev nodes or other nodes that cluster towards the endpoints, the Leja ordering is thought to be close to optimal. For this ordering, we compute the pick the point that maximizes the product of the distances with the points we have selected previously. That is:

$$xnew_0 = \max_j |x_j|$$
$$xnew_j = x\left[\text{argmax} \prod_{i=1}^{j-1} |x_j - xnew_i|\right]$$

Newton with these orderings is much more stable than following the sequential order (it is still a bit less stable than Barycentric Lagrange, which we explain below).

2.3 Barycentric Lagrange interpolation

Some common objections to the use of the Lagrange basis, as we have described it, include:

- Each evaluation, if we don't store the Lagrange basis, is $O(n^2)$.
- Adding a new point requires new computation
- Computing the Lagrange polynomials for large n can be unstable.

These are typically contrasted with Newton, which has the following attributes:

- Finding the coefficients is $O(n^2)$ and may be unstable for large n.
- Evaluation is efficient and stable: $O(n)$ per point using Horner
- Adding a new point requires little work ($O(n)$), Newton lends itself to updates.

It is, however, possible to greatly improve Lagrange interpolation and address all of the issues listed above. This improved Lagrange method, known as Barycentric Lagrange, is an incredibly competitive and stable way to compute, update and evaluate the polynomial interpolant.

We will derive two so-called *barycentric formulas*. Each will lead to a representation of the interpolant and a corresponding evaluation algorithm. The one we will suggest using and that has all the properties listed above is the *second barycentric formula*.

2.3.1 First Barycentric Lagrange formula

Consider the monic polynomial of degree $n+1$ defined by $\Psi(x) = \prod_{j=0}^{n}(x - x_j)$. We note the following relationship to the Lagrange polynomials:

$$L_j(x) = \left(\frac{1}{\prod_{i \neq j}(x_j - x_i)} \right) = \omega_j \frac{\Psi(x)}{x - x_j} \tag{2.7}$$

Where we have denoted the constant $\omega_j = \frac{1}{\prod_{i \neq j}(x_j - x_i)}$. We further note that $\Psi'(x_j) = \prod_{i \neq j}(x_j - x_i)$ (show this with a chain rule, all terms vanish except this one), and so $\omega_j = \frac{1}{\Psi'(x_j)}$.

But then, the polynomial interpolant can be written as:

$$p(x) = \sum_{j=0}^{n} y_j L_j(x) = \Psi(x) \sum_{j=0}^{n} y_j \frac{\omega_j}{x - x_j} \tag{2.8}$$

We call Equation 2.8 the **first barycentric formula**. This formula suggests the following algorithm:

1. Compute ω_j (only dependent on x_j). For each target point in $\mathbf{z}$, evaluate $\Psi(z)$.
2. Evaluation is $O(n)$ per z_k after that, using the *first barycentric formula*.

We note that even in this format, updates are cheap, since we only need to multiply the weights ω_j and $\Psi(z)$ by one factor. We can also update our evaluations cheaply.

2.3.2 Second Barycentric Lagrange formula

One of the issues remaining that motivate modifying the first formula is that the weights ω_j can grow quite a bit, especially for equispaced points; this can lead to instability. The second formula gies us a way to both reduce the growth of these weights *and* in fact remove the need to evaluate $\Psi(x)$ altogether. To do this, we first observe that given any interpolation routine, there's always a way to write 1 (and we can always divide by 1!):

$$1 = \sum_{j=0}^{n} L_j(x) = \Psi(x) \sum_{j=0}^{n} \frac{\omega_j}{x - x_j} \tag{2.9}$$

We say that the Lagrange polynomials are a "partition of unity" because of this. So, if we divide the first barycentric formula by this:

$$p(x) = \frac{\Psi(x) \sum_{j=0}^{n} y_j \frac{\omega_j}{x-x_j}}{\Psi(x) \sum_{j=0}^{n} \frac{\omega_j}{x-x_j}} \tag{2.10}$$

$$p(x) = \frac{\sum_{j=0}^{n} y_j \frac{\omega_j}{x-x_j}}{\sum_{j=0}^{n} \frac{\omega_j}{x-x_j}} \tag{2.11}$$

This is the *second barycentric formula.* We note that the $\Psi(x)$ cancel, and further, that because the weights appear in both the numerator and the denominator, we can scale all weights ω_j by the same constant factor, and the formula still works.

For a given set of interpolation points, say equispaced, Chebyshev points or others (e.g. Gauss-Legendre points, which we will see in the context of numerical integration), we can derive formulas for these weights. We can then remove any constant factor that shows up in all weights, making evaluation even more stable (and requiring less work). For example:

1. **Equispaced points** with spacing h: The weights are $\omega_j = (-1)^j \binom{n}{j} \frac{1}{h^n n!}$. We can remove the constant factor to get the normalized weights $\tilde{\omega}_j = (-1)^j \binom{n}{j}$.
2. **Chebyshev points of the first kind:** After normalization, the weights are all in $[-1, 1]$: $\tilde{\omega}_j = (-1)^j \sin\left(\frac{(2j-1)\pi}{2n+2}\right)$.
3. **Chebyshev points of the second kind:** After normalization, the weights are all ± 1 or $\pm 1/2$: $\tilde{\omega}_j = (-1)^j \eta_j$ where $\eta_j = 1$ for $j = 1, \ldots, n-1$ and $\eta_0 = \eta_n = 1/2$.

Another benefit of the second formula is that even though these weights are calculated for the interval $[-1, 1]$, the normalized weights work for any interval $[a, b]$ (for the first formula, you would have to multiply them by $2^n (b-a)^{-n}$).

2.3.3 Evaluation algorithm:

1. For a given set of interpolation points, compute ω_j (or use a formula if available).
2. For each target, evaluate $\frac{\omega_j}{z-x_j}$. Use the second barycentric formula to evaluate $p(z)$ ($O(n)$, more stable).

2.4 Interpolation Error Analysis

2.4.1 Mathematical preliminaries

Definition 5 *Let $\mathcal{I}[x_0, x_1, \ldots, x_n]$ denote the smallest interval that contains $\{x_j\}_{j=0}^{n}$.*

2.4.2 Polynomial interpolation error

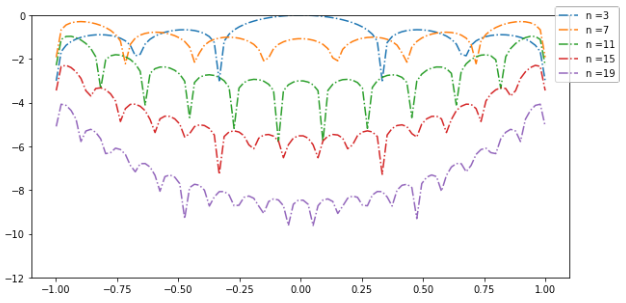

Figure 2.4: Log interpolation error for interpolant in equispaced nodes. Note that error goes down as we increase n, but not uniformly (the error curves have a smile shape)

This is the main result in this section: it gives us a bound for the evaluation error between the true function and the interpolant:

Theorem 2.1 (Polynomial interpolation error) *Let $\{x_j\}_{j=0}^{n}$ distinct interpolation locations, and $f \in C^{n+1}(\mathcal{I}_t)$ where $\mathcal{I}_t = \mathcal{I}[x_0, x_1, \ldots, x_n, t]$. Then, there exists $\alpha \in \mathcal{I}_t$ such that the interpolation error:*

$$E(t) = f(t) - \sum_{j=0}^{n} f(x_j)L_j(t) = \frac{f^{n+1}(\alpha)}{(n+1)!}\prod_{j=0}^{n}(x - x_j) = \frac{f^{n+1}(\alpha)}{(n+1)!}\Psi(t) \tag{2.12}$$

This gives us a bound for all $t \in \mathcal{I}[x_0, x_1, \ldots, x_n]$:

$$|E(t)| \leq \frac{\|f^{n+1}\|_\infty}{(n+1)!}|\Psi(t)| \tag{2.13}$$

We note that this theorem assumes our function is $n+1$ times continuously differentiable on the interval where our interpolation locations lie. Also, the error bound we get has two factors: one that purely depends on the magnitude of the $n+1$ derivatives of the function, and another one that purely depends on the choice of interpolation locations.

We dont include the proof for this theorem, but note that it involves applying Rolle's theorem (a special case of the Mean Value Theorem) to a special function $n+1$ times.

2.4.3 Sensitivity to errors in the data

This section was not discussed in class, but it is included for the curious reader. Say our interpolation data is slightly perturbed (this is at least true because of finite precision representation):

$$y_j = f(x_j) + \varepsilon_j$$

Then, instead of the true interpolant, we would compute:

$$p_\varepsilon(x) = \sum_{j=0}^{n} (f(x_j) + \varepsilon_j) L_j(x)$$

Then, the difference between the true solution $p(x)$ and the exact solution to the perturbed data $p_\varepsilon(x)$ is:

$$\begin{aligned}
\|p - p_\varepsilon\|_\infty &= \max_{x \in I_t} \left| \sum_{j=0}^{n} \varepsilon_j L_j(x) \right| \\
&= \max_{x \in I_t} \sum_{j=0}^{n} |\varepsilon_j| |L_j(x)| \\
&= \|\boldsymbol{\varepsilon}\|_\infty \left(\max_{x \in I_t} \sum_{j=0}^{n} |L_j(x)| \right) \\
&= \|\boldsymbol{\varepsilon}\|_\infty \Lambda_n(\{x_j\}_{j=0}^{n})
\end{aligned}$$

where $\Lambda_n(\{x_j\}_{j=0}^{n}) = \max_{x \in I_t} \sum_{j=0}^{n} |L_j(x)|$ is known as the **Lebesgue constant**, and it depends only on the distribution of the interpolation nodes.

We observe that both the interpolation error and the estimate on sensitivity to perturbations in the data (and so, numerical stability) of polynomial interpolation are both bounded by a product of two factors, one of which purely depends on the distribution of points.

2.5 Equispaced points and the Runge phenomenon

When we look at $|\Psi(x)|$ for increasing n, we see that for equispaced points, our error bound tends to increase in a thin boundary layer close to the interval boundaries. If this is compounded with high values for $f^{n+1}(x)$, the interpolation error can also display this behavior. This is known as the *Runge phenomena.* The key issue in interpolation with equispaced nodes is that convergence is pointwise, but the rate of convergence is very heterogeneous, and the interpolants can have horrible growth near the boundaries.

When we look at the Lebesgue constant for equispaced nodes, we get that $\Lambda_n = \frac{2^n}{en \log n}$. Which means interpolation for equispaced nodes may be unstable for large n.

The best way to see there is an issue with equispaced nodes is to test them with a function for which they are problematic. Historically, the "Runge function" $f(x) = \frac{1}{1+25x^2}$ was used to showcase this (Figure 2.5).

When we try to interpolate it with equispaced points and look at the log error, we see something very concerning near the interval boundaries. And this only gets worse as we increase n, as we can see in Figure 2.6.

Both of these issues motivate finding a better set of interpolation nodes, and in some sense, it tells us what we can control (what the optimal nodes should accomplish): minimizing $\|\Psi\|_\infty$ and the Lebesgue constant. In this sense, the set of *Chebyshev nodes* can be shown to be near-optimal. We will derive some of these results later, when we discuss function approximation. However, let us describe what the Chebyshev nodes are, why they work and test if they get rid of the Runge phenomenon.

There are really four kinds of Chebyshev nodes: each kind corresponds to the roots of a family of orthogonal polynomials. For now, what we can say is that they all have the property that their

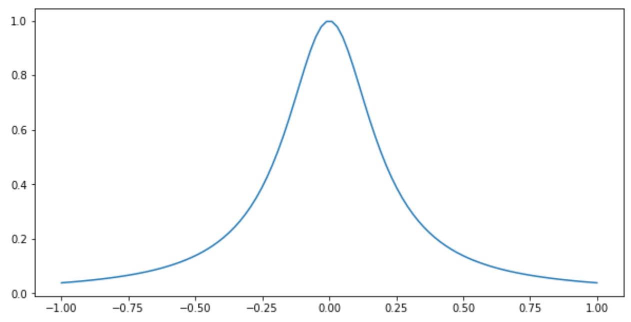

Figure 2.5: The infamous and inoffensive looking Runge function

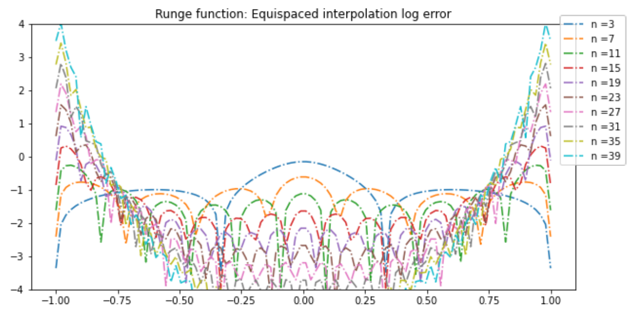

Figure 2.6: Log interpolation error for the Runge function on equispaced nodes

distribution corresponds to taking equi-spaced nodes on the unit circle, and then projecting those nodes onto the x-axis (Figure 2.7). The Chebyshev nodes of the first kind, for instance, can be obtained from the formula:

$$x_k = \cos\left(\frac{(2k+1)\pi}{2n+2}\right) \quad k = 0, \dots, n \tag{2.14}$$

Chebyshev nodes of the second kind are similarly defined as:

$$x_k = \cos\left(\frac{k\pi}{n}\right) \quad k = 0, \dots, n \tag{2.15}$$

We note that these include the endpoints of $[-1, 1]$ (whereas those of the first kind are inside $(-1, 1)$).

The fact that we are sampling uniformly on the unit circle causes the points to cluster near the endpoints, with a probability density $g(x) = \frac{2}{\pi}\sqrt{1-x^2}$. We can say, for instance, that for $n+1$ Chebyshev nodes, the distance between points near the endpoints is proportional to $1/n^2$ (as opposed to $1/n$). And we can show this clustering at the endpoints is exactly what is needed to get rid of the Runge phenomenon, and to get *uniform convergence of the polynomial interpolant error.*

When we use Chebyshev nodes with the Runge function, we see a big difference (Figure 2.8).

There's a few ways we can see what the big differences between equispaced and Chebyshev nodes are. We can visualize the Lagrange basis, and see that it has high oscillations for equispaced, but is between -1 and 1 for Chebyshev. This tells us that the Lebesgue constant for Chebyshev must be way better than that for Equispaced; in fact, it can be shown that $\Lambda_n \sim \frac{2}{\pi}\log(n+1) + 1$ for Chebyshev. This is very close to the proven lower bound of $\Lambda_n \geq \frac{2}{\pi}\log(n+1) + 0.5215$ (for all sets of $n+1$ points).

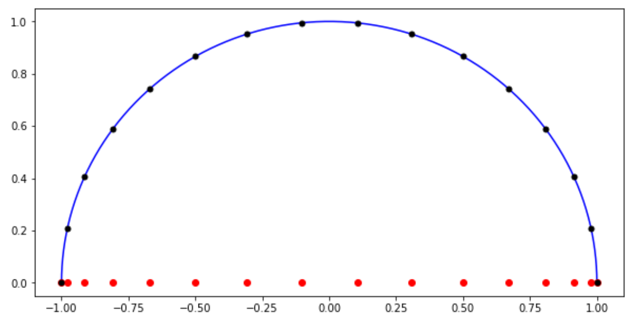

Figure 2.7: Chebyshev nodes of the first kind (in red) obtained from x coordinates in the unit circle. Observe how the points cluster near the endpoints.

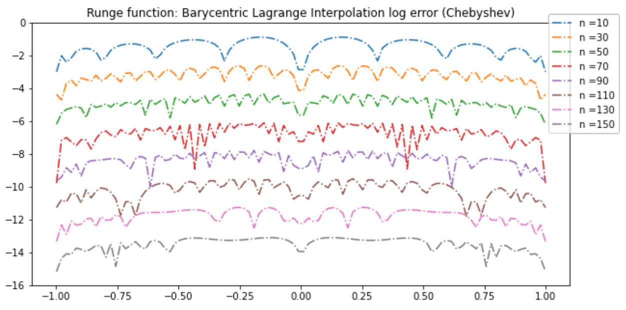

Figure 2.8: Log plot for barycentric lagrange interpolation of the Runge function on Chebyshev nodes. Notice the curves are now practically flat, and uniformly go to zero as n increases.

We can also directly plot $\Psi(x)$ for both sets of points, as is done in Figure 2.9. By comparing $\log_{10} |\Psi(x)|$ for both point distributions, we see the same pattern that shows up in the interpolation error: for equispaced, the error is orders of magnitude different than in the center. For Chebyshev, it is uniformly controlled.

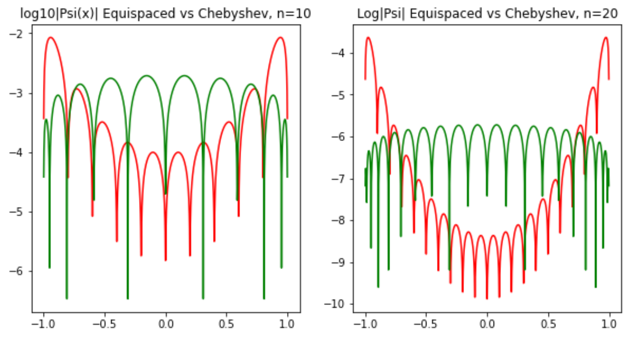

Figure 2.9: Plot of $\log_{10} |\Psi(x)|$ for Equispaced (red) vs Chebyshev nodes (green). We can see that these look exactly like the interpolation error, and the difference between them becomes starker for higher n.

2.6 Hermite interpolation

We can generalize the idea of a function interpolant to involve not only matching function evaluations, but as is the case for Taylor expansions, to match derivative evaluations (and most generally, linear functionals of $f(x)$). *Hermite polynomial interpolation* is a particular case: We ask for $p \in \mathcal{P}_{2n+1}$ such that it satisfies the $2n+2$ conditions:

$$\begin{aligned} p(x_j) = f(x_j) = y_j \quad j = 0, \ldots, n \\ p'(x_j) = f'(x_j) = z_j \quad j = 0, \ldots, n \end{aligned}$$

We can generalize all of the ideas for polynomial evaluation to tackle this problem. First: using a polynomial basis for $\mathcal{P}_{2n+1}$, we can define what is known as the *Confluent Vandermonde* matrix; each row of this matrix corresponds to satisfying one of the $2n+2$ interpolation conditions; each column evaluates each basis term and its derivative at all interpolation nodes.

We can also come up with *Lagrange-Hermite* and *Lagrange-Newton* bases of $\mathcal{P}_{2n+1}$, leading to similar methods to represent and compute the Hermite interpolant. Finally, we can derive an equivalent error bound for the Hermite interpolation error.

First, let us discuss Hermite-Lagrange. The idea is to generate a basis such that one of the conditions is equal to one, the rest is equal to zero. We define two kinds of basis terms $H_j, K_j \in \mathcal{P}_{2n+1}$ such that:

$$\begin{aligned} H_j(x_i) = \delta_{i,j} \qquad H_j'(x_i) = 0 \quad i = 0, 1, \ldots, n \\ K_j(x_i) = 0 \qquad K_j'(x_i) = \delta_{i,j} \quad i = 0, 1, \ldots, n \end{aligned}$$

Once we have this basis, we can immediately represent the Hermite interpolant as:

$$p(x) = \sum_{j=0}^{n} f(x_j) H_j(x) + f'(x_j) K_j(x) \tag{2.16}$$

It turns out these Hermite-Lagrange polynomials can be written in terms of the regular Lagrange basis. We have the following formulas:

$$H_j(x) = (1 - 2\eta_j(x - x_j)) \left(L_j(x)\right)^2 \tag{2.17}$$
$$K_j(x) = (x - x_j) \left(L_j(x)\right)^2 \tag{2.18}$$

with $\eta_j = L_j'(x_j)$. To show that these polynomials satisfy the conditions we asked for, a hint is to look at the polynomial $M_j(x) = (L_j(x))^2 \in \mathcal{P}_{\in\backslash}$. Show that $M_j(x_i) = \delta_{i,j}$ and $M_j'(x_i) = 2\eta_j(x_j)\delta_{i,j}$. This can be then used to show H_j and K_j satisfy the conditions set for the Hermite-Lagrange basis.

Theorem 2.2 *Let $n \geq 0$, given Hermite interpolation data, the Hermite interpolant is unique.*

This can be shown either via the determinant of the confluent Vandermonde matrix, or using the Hermite-Lagrange basis.

2.6.1 Newton-Hermite basis and coefficients

We can define the Newton-Hermite basis using similar reasoning as was used for Newton: We want a basis that sequentially takes care of the interpolation conditions for each point, starting at x_0. This makes the confluent Vandermonde matrix triangular. The basis terms are defined as:

$$\begin{aligned}
\mu_0(x) &= 1 \\
\mu_1(x) &= (x - x_0) \\
\mu_2(x) &= (x - x_0)^2 \\
\mu_3(x) &= (x - x_0)^2(x - x_1) \\
\mu_4(x) &= (x - x_0)^2(x - x_1)^2 \\
&\vdots \\
\mu_{2n+1} &= (x - x_n)\prod_{j=1}^{n-1}(x - x_j)^2
\end{aligned}$$

That is, we add simple and then double roots at each additional node. The coefficients can be expressed in terms of divided differences, except we must add the following notation:

$$f[x_i, x_i] = f'(x_i) \tag{2.19}$$

That is, we are going to allow repeated nodes in our difference formulas, and define differences of higher order recursively as before. For example, this means that:

$$f[x_i, x_i, x_{i+1}] = \frac{f[x_i, x_{i+1}] - f[x_i, x_i]}{x_{i+1} - x_i} = \frac{f[x_i, x_{i+1}] - f'(x_i)}{x_{i+1} - x_i} \tag{2.20}$$

The $2n + 2$ coefficients can be recovered from the Hermite-Newton tableau (table):

$$\begin{bmatrix}
f(x_0) & f[x_0, x_0] & f[x_0, x_0, x_1] & \cdots & f[x_0, x_0, x_1, x_1, \cdots, x_n, x_n] \\
f(x_0) & f[x_0, x_1] & f[x_0, x_1, x_1] & \cdots & \\
f(x_1) & f[x_1, x_1] & f[x_1, x_1, x_2] & \iddots & \\
f(x_1) & f[x_1, x_2] & f[x_1, x_2, x_2] & \iddots & \\
\vdots & \vdots & \iddots & & \\
f(x_{n-1}) & f[x_{n-1}, x_{n-1}] & f[x_{n-1}, x_{n-1}, x_n] & & \\
f(x_{n-1}) & f[x_{n-1}, x_n] & f[x_{n-1}, x_n, x_n] & & \\
f(x_n) & f[x_n, x_n] & & & \\
f(x_n) & & & &
\end{bmatrix} \tag{2.21}$$

Each new interpolation node requires the addition of two new Newton basis terms, and computing two anti-diagonals in the tableau. The $2n + 2$ coefficients are once again located at the top row of the tableau.

2.6.2 Hermite Error Analysis

Theorem 2.3 (Hermite polynomial interpolation error) *Let $\{x_j\}_{j=0}^{n}$ distinct interpolation locations, and $f \in C^{2n+2}(\mathcal{I}_t)$ where $\mathcal{I}_t = \mathcal{I}[x_0, x_1, \ldots, x_n, t]$. Then, there exists $\alpha \in \mathcal{I}_t$ such that the Hermite interpolation error:*

$$E_H(t) = f(t) - \sum_{j=0}^{n} f(x_j)H_j(t) + f'(x_j)K_j(x) = \frac{f^{2n+2}(\alpha)}{(2n+2)!}\left(\Psi(t)\right)^2 \tag{2.22}$$

This gives us a bound for all $t \in \mathcal{I}[x_0, x_1, \ldots, x_n]$:

$$|E_H(t)| \leq \frac{\|f^{2n+2}\|_\infty}{(2n+2)!}|\Psi(t)|^2 \tag{2.23}$$

We note two important things:

- The error for the Hermite interpolant at $n+1$ points looks similar to that for the standard polynomial interpolant using data at $2n+2$ points.
- The error also has a product of two terms, one of which depends on the interpolation points and *is the same as that for the standard polynomial interpolation, squared.* This tells us Hermite might suffer from the same Runge phenomenon if we used equispaced points and moderately large n. It also tells us Chebyshev nodes will also be great and provide uniform error convergence for Hermite interpolation.

2.6.3 Bonus: Hermite Interpolation and Barycentric Lagrange

As a bonus, we can investigate whether the approach we took to improve vanilla Lagrange interpolation would also work for the Hermite case. Recall that both types of basis polynomials H_j, K_j involved the square of the Lagrange polynomial, denoted $M_j(x) = L_j(x)^2$. If we factor $L_j(x)$ as we did in the barycentric Lagrange section, we get:

$$M_j(x) = \Psi(x)^2\omega_j^2\frac{1}{(x-x_j)^2}$$

Plugging this into the definitions of our basis functions,

$$H_j(x) = \Psi(x)^2\omega_j^2\frac{(1-2\eta_j(x-x_j))}{(x-x_j)^2}$$
$$K_j(x) = \Psi(x)^2\omega_j^2\frac{1}{(x-x_j)}$$

And so, the Hermite interpolant $h(x)$ can be written as:

$$h(x) = \Psi(x)^2\sum_{j=0}^{n}\omega_j^2\left[y_j\frac{(1-2\eta_j(x-x_j))}{(x-x_j)^2} + z_j\frac{1}{(x-x_j)}\right]$$

which would be the *first barycentric formula.* We can then divide by the representation of the function equal to 1 (which means $y_j = 1, z_j = 0$), to get:

$$h(x) = \frac{\sum_{j=0}^{n}\omega_j^2\left[y_j\frac{(1-2\eta_j(x-x_j))}{(x-x_j)^2} + z_j\frac{1}{(x-x_j)}\right]}{\sum_{j=0}^{n}\omega_j^2\left[\frac{(1-2\eta_j(x-x_j))}{(x-x_j)^2}\right]}$$

which is the *Second barycentric formula.* We can now quickly modify our code for Barycentric Lagrange to produce a Barycentric Hermite-Lagrange interpolation code!