9 Polynomial Interpolation

Lab Objective: *Learn and compare three methods of polynomial interpolation: standard Lagrange interpolation, Barycentric Lagrange interpolation and Chebyshev interpolation. Explore Runge's phenomenon and how the choice of interpolating points affect the results. Use polynomial interpolation to study air polution by approximating graphs of particulates in air.*

Polynomial Interpolation

Polynomial interpolation is the method of finding a polynomial that matches a function at specific points in its range. More precisely, if $f(x)$ is a function on the interval $[a, b]$ and $p(x)$ is a polynomial then $p(x)$ interpolates the function $f(x)$ at the points $x_0, x_1, \dots, x_n$ if $p(x_j) = f(x_j)$ for all $j = 0, 1, \dots, n$. In this lab most of the discussion is focused on using interpolation as a means of approximating functions or data, however, polynomial interpolation is useful in a much wider array of applications.

Given a function $f(x)$ and a set of unique points $\{x_i\}_{i=0}^n$, it can be shown that there exists a unique interpolating polynomial $p(x)$. That is, there is one and only one polynomial of degree n that interpolates $f(x)$ through those points. This uniqueness property is why, for the remainder of this lab, an interpolating polynomial is referred to as *the* interpolating polynomial. One approach to finding the unique interpolating polynomial of degree n is Lagrange interpolation.

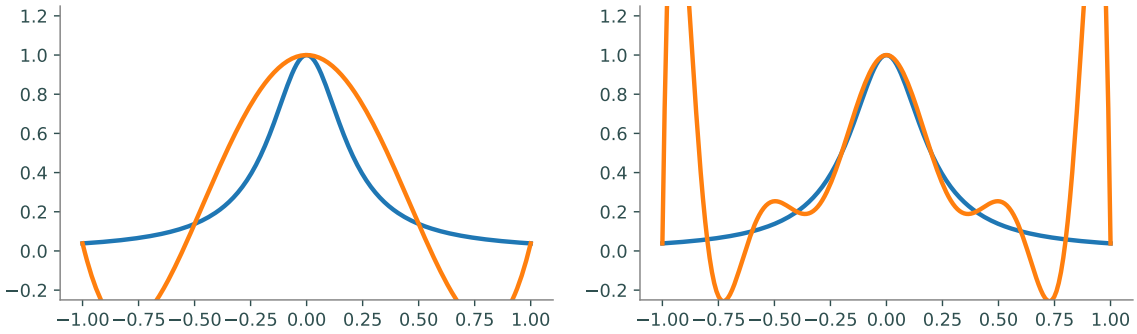
Lagrange interpolation

Given a set $\{x_i\}_{i=1}^n$ of n points to interpolate, a family of n basis functions with the following property is constructed:

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

The Lagrange form of this family of basis functions is

$$L_j(x) = \frac{\prod_{k=1, k\neq j}^{n} (x - x_k)}{\prod_{k=1, k\neq j}^{n} (x_j - x_k)} \tag{9.1}$$



(a) Interpolation using 5 equally spaced points.

(b) Interpolation using 11 equally spaced points.

Figure 9.1: Interpolations of Runge's function $f(x) = \frac{1}{1+25x^2}$ with equally spaced interpolating points.

Each of these Lagrange basis functions is a polynomial of degree $n-1$ and has the necessary properties as given above.

Problem 1. Define a function `lagrange()` that will be used to construct and evaluate an interpolating polynomial on a domain of x values. The function should accept two NumPy arrays of length n which contain the x and y values of the interpolating points as well as a NumPy array of values of length m at which the interpolating polynomial will be evaluated.

Within `lagrange()`, write a subroutine that will evaluate each of the n Lagrange basis functions at every point in the domain. It may be helpful to follow these steps:

1. Compute the denominator of each L_j (as in Equation 9.1) .
2. Using the previous step, evaluate L_j at all points in the computational domain (this will give you m values for each L_j.)
3. Combine the results into an $n \times m$ NumPy array, consisting of each of the n L_j evaluated at each of the m points in the domain.

You may find the functions `np.product()` and `np.delete()` to be useful while writing this method.

Lagrange interpolation is completed by combining the Lagrange basis functions with the y-values of the function to be interpolated $y_i = f(x_i)$ in the following manner:

$$p(x) = \sum_{j=1}^{n} y_j L_j(x) \tag{9.2}$$

This will create the unique interpolating polynomial.

Since polynomials are typically represented in their expanded form with coefficients on each of the terms, it may seem like the best option when working with polynomials would be to use Sympy, or NumPy's `poly1d` class to compute the coefficients of the interpolating polynomial individually. This is rarely the best approach, however, since expanding out the large polynomials that are required can quickly lead to instability (especially when using large numbers of interpolating points). Instead, it is usually best just to leave the polynomials in unexpanded form (which is still a polynomial, just not a pretty-looking one), and compute values of the polynomial directly from this unexpanded form.

```
# Evaluate the polynomial (x-2)(x+1) at 10 points without expanding the ↩
    expression.
>>> pts = np.arange(10)
>>> (pts - 2) * (pts + 1)
array([ 2,  0,  0,  2,  6, 12, 20, 30, 42, 56])
```

In the given example, there would have been no instability if the expression had actually been expanded but in the case of a large polynomial, stability issues can dominate the computation. Although the coefficients of the interpolating polynomials will not be explicitly computed in this lab, polynomials are still being used, albeit in a different form.

Problem 2. Complete the implementation of `lagrange()`.

Evaluate the interpolating polynomial at each point in the domain by combining the y values of the interpolation points and the evaluated Lagrange basis functions from Problem 1 as in Equation 9.2. Return the final array of length m that consists of the interpolating polynomial evaluated at each point in the domain.

You can test your function by plotting Runge's function $f(x) = \frac{1}{1+25x^2}$ and your interpolating polynomial on the same plot for different values of n equally spaced interpolating values then comparing your plot to the plots given in Figure 9.1.

The Lagrange form of polynomial interpolation is useful in some theoretical contexts and is easier to understand than other methods, however, it has some serious drawbacks that prevent it from being a useful method of interpolation. First, Lagrange interpolation is $O(n^2)$ where other interpolation methods are $O(n^2)$ (or faster) at startup but only $O(n)$ at run-time, Second, Lagrange interpolation is an unstable algorithm which causes it to return innacurate answers when larger numbers of interpolating points are used. Thus, while useful in some situations, Lagrange interpolation is not desirable in most instances.

Barycentric Lagrange interpolation

Barycentric Lagrange interpolation is simple variant of Lagrange interpolation that performs much better than plain Lagrange interpolation. It is essentially just a rearrangement of the order of operations in Lagrange multiplication which results in vastly improved perfomance, both in speed and stability.

Barycentric Lagrange interpolation relies on the observation that each basis function L_j can be rewritten as

$$L_j(x) = \frac{w(x)}{(x - x_j)} w_j$$

where

$$w(x) = \prod_{j=1}^{n}(x - x_j)$$

and

$$w_j = \frac{1}{\prod_{k=1,k\neq j}^{n}(x_j - x_k)}.$$

The w_j's are known as the *barycentric weights*.

Using the previous equations, the interpolating polynomial can be rewritten

$$p(x) = w(x)\sum_{j=1}^{n}\frac{w_j y_j}{x - x_j}$$

which is the *first barycentric form*. The computation of $w(x)$ can be avoided by first noting that

$$1 = w(x)\sum_{j=1}^{n}\frac{w_j}{x - x_j}$$

which allows the interpolating polynomial to be rewriten as

$$p(x) = \frac{\sum_{j=1}^{n}\frac{w_j y_j}{x - x_j}}{\sum_{j=1}^{n}\frac{w_j}{x - x_j}}$$

This form of the Lagrange interpolant is known as the *second barycentric form* which is the form used in Barycentric Lagrange interpolation. So far, the changes made to Lagrange interpolation have resulted in an algorithm that is $O(n)$ once the barycentric weights (w_j) are known. The following adjustments will improve the algorithm so that it is numerically stable and later discussions will allow for the quick addition of new interpolating points after startup.

The second barycentric form makes it clear that any factors that are common to the w_k can be ignored (since they will show up in both the numerator and denominator). This allows for an important improvement to the formula that will prevent overflow error in the arithmetic. When computing the barycentric weights, each element of the product $\prod_{k=1,k\neq j}^{n}(x_j - x_k)$ should be multiplied by C^{-1}, where $4C$ is the width of the interval being interpolated (C is known as the *capacity* of the interval). In effect, this scales each barycentric weight by C^{1-n} which helps to prevent overflow during computation. Thus, the new barycentric weights are given by

$$w_j = \frac{1}{\prod_{k=1,k\neq j}^{n}[(x_j - x_k)/C]}.$$

Once again, this change is possible since the extra factor C^{1-n} is cancelled out in the final product. This process is summed up in the following code:

```
# Given a NumPy array xint of interpolating x-values, calculate the weights.
>>> n = len(xint)                    # Number of interpolating points.
>>> w = np.ones(n)                   # Array for storing barycentric weights.
# Calculate the capacity of the interval.
>>> C = (np.max(xint) - np.min(xint)) / 4
```

```
>>> shuffle = np.random.permutation(n-1)
>>> for j in range(n):
>>>     temp = (xint[j] - np.delete(xint, j)) / C
>>>     temp = temp[shuffle]        # Randomize order of product.
>>>     w[j] /= np.product(temp)
```

The order of `temp` was randomized so that the arithmetic does not overflow due to poor ordering (if standard ordering is used, overflow errors can be encountered since all of the points of similar magnitude are multiplied together at once). When these two fixes are combined, the Barycentric Algorithm becomes numerically stable.

Problem 3. Create a class that performs Barycentric Lagrange interpolation. The constructor of your class should accept two NumPy arrays which contain the x and y values of the interpolation points. Store these arrays as attributes. In the constructor, compute the corresponding barycentric weights and store the resulting array as a class attribute. Be sure that the relative ordering of the arrays remains unchanged.

Implement the `__call__()` method so that it accepts a NumPy array of values at which to evaluate the interpolating polynomial and returns an array of the evaluated points. Your class can be tested in the same way as the Lagrange function written in Problem 2

ACHTUNG!

As currently explained and implemented, the Barycentric class from Problem 3 will fail when a point to be evaluated exactly matches one of the x-values of the interpolating points. This happens because a divide by zero error is encountered in the final step of the algorithm. The fix for this, although not required here, is quite easy: keep track of any problem points and replace the final computed value with the corresponding y-value (since this is a point that is exactly interpolated). If you do not implement this fix, just be sure not to pass in any points that exactly match your interpolating values.

Another advantage of the barycentric method is that it allows for the addition of new interpolating points in $O(n)$ time. Given a set of existing barycentric weights $\{w_j\}_{j=1}^n$ and a new interpolating point x_i, the new barycentric weight is given by

$$w_i = \frac{1}{\prod_{k=1}^{n}(x_i - x_k)}.$$

In addition to calculating the new barycentric weight, all existing weights should be updated as follows $w_j = \frac{w_j}{x_j - x_i}$.

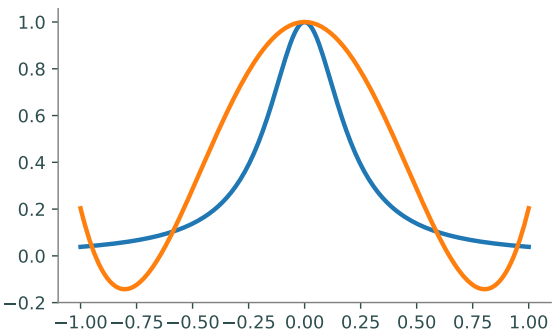
Problem 4. Include a method in the class written in Problem 3 that allows for the addition of new interpolating points by updating the barycentric weights. Your function should accept two NumPy arrays which contain the x and y values of the new interpolation points. Update and store the old weights then extend the class attribute arrays that store the weights, and the x and y values of the interpolation points with the new data. When updating all class attributes, make sure to maintain the same relative order.

The implementation outlined here calls for the y-values of the interpolating points to be known during startup, however, these values are not needed until run-time This allows the y-values to be changed without having to recompute the barycentric weights. This is an additional advantage of Barycentric Lagrange interpolation.

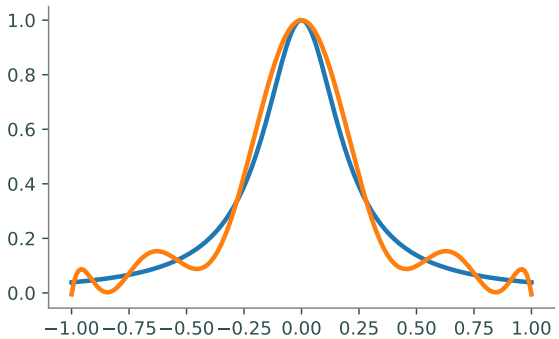
Scipy's Barycentric Lagrange class

Scipy includes a Barycentric interpolator class. This class includes the same functionality as the class described in Problems 3 and 4 in addition to the ability to update the y-values of the interpolation points. The following code will produce a figure similar to Figure 9.1b.

```
>>> from scipy.interpolate import BarycentricInterpolator

>>> f = lambda x: 1/(1+25 * x**2)   # Function to be interpolated.
# Obtain the Chebyshev extremal points on [-1,1].
>>> n = 11
>>> pts = np.linspace(-1, 1, n)
>>> domain = np.linspace(-1, 1, 200)

>>> poly = BarycentricInterpolator(pts[:-1])
>>> poly.add_xi(pts[-1])          # Oops, forgot one of the points.
>>> poly.set_yi(f(pts))           # Set the y values.

>>> plt.plot(domain, f(domain))
>>> plt.plot(domain, poly.eval(domain))
```



(a) Polynomial using 5 Chebyshev roots.



(b) Polynomial using 11 Chebyshev roots.

Figure 9.2: Example of overcoming Runge's phenomenon by using Chebyshev nodes for interpolating values. Plots made using Runge's function $f(x) = \frac{1}{1+25x^2}$. Compare with Figure 9.1

Chebyshev Interpolation

Chebyshev Nodes

As has been mentioned previously, the Barycentric version of Lagrange interpolation is a stable process that does not accumulate large errors, even with extreme inputs. However, polynomial interpolation itself is, in general, an ill-conditioned problem. Thus, even small changes in the interpolating values can give drastically different interpolating polynomials. In fact, poorly chosen interpolating points can result in a very bad approximation of a function. As more points are added, this approximation can worsen. This increase in error is called *Runge's phenomenon*.

The set of equally spaced points is an example of a set of points that may seem like a reasonable choice for interpolation but in reality produce very poor results. Figure 9.1 gives an example of this using Runge's function. As the number of interpolating points increases, the quality of the approximation deteriorates, especially near the endpoints.

Although polynomial interpolation has a great deal of potential error, a good set of interpolating points can result in fast convergence to the original function as the number of interpolating points is increased. One such set of points is the Chebyshev extremal points which are related to the Chebyshev polynomials (to be discussed shortly). The $n+1$ Chebyshev extremal points on the interval $[a,b]$ are given by the formula $y_j = \frac{1}{2}(a+b+(b-a)\cos(\frac{j\pi}{n}))$ for $j = 0, 1, \ldots, n$. These points are shown in Figure 9.3. One important feature of these points is that they are clustered near the endpoints of the interval, this is key to preventing Runge's phenomenon.

Problem 5. Write a function that defines a domain `x` of 400 equally spaced points on the interval $[-1,1]$. For $n = 2^2, 2^3, \ldots, 2^8$, repeat the following experiment.

1. Interpolate Runge's function $f(x) = 1/(1+25x^2)$ with n equally spaced points over $[-1,1]$ using SciPy's `BarycentricInterpolator` class, resulting in an approximating function $\tilde{f}$. Compute the absolute error $\|f(\mathbf{x}) - \tilde{f}(\mathbf{x})\|_\infty$ of the approximation using `la.norm()` with `ord=np.inf`.

2. Interpolate Runge's function with $n+1$ Chebyshev extremal points, also via SciPy, and compute the absolute error.

Plot the errors of each method against the number of interpolating points n in a log-log plot.

To verify that your figure make sense, try plotting the interpolating polynomials with the original function for a few of the larger values of n.

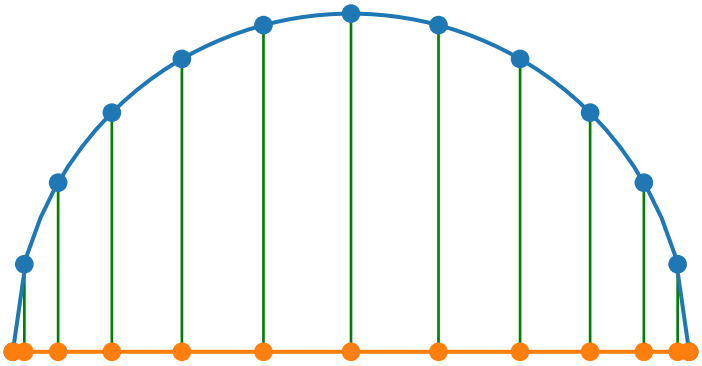


Figure 9.3: The Chebyshev extremal points. The n points where the Chebyshev polynomial of degree n reaches its local extrema. These points are also the projection onto the x-axis of n equally spaced points around the unit circle.

Chebyshev Polynomials

The Chebyshev roots and Chebyshev extremal points are closely related to a set of polynomials known as the Chebyshev polynomials. The first two Chebyshev polynomials are defined as $T_0(x) = 1$ and $T_1(x) = x$. The remaining polynomials are defined by the recursive algorithm $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$. The Chebyshev polynomials form a complete basis for the polynomials in $\mathbb{R}$ which means that for any polynomial $p(x)$, there exists a set of unique coefficients $\{a_k\}_{k=0}^n$ such that

$$p(x) = \sum_{k=0}^{n} a_k T_k.$$

Finding the Chebyshev representation of an interpolating polynomial is a slow process (dominated by matrix multiplication or solving a linear system), but when the interpolating values are the Chebyshev extrema, there exists a fast algorithm for computing the Chebyshev coefficients of the interpolating polynomial. This algorithm is based on the Fast Fourier transform which has temporal complexity $O(n \log n)$. Given the $n+1$ Chebyshev extremal points $y_j = \cos(\frac{j\pi}{n})$ for $j = 0, 1, \ldots, n$ and a function f, the unique n-degree interpolating polynomial $p(x)$ is given by

$$p(x) = \sum_{k=0}^{n} a_k T_k$$

where

$$a_k = \gamma_k \Re \left[DFT(f(y_0), f(y_1), \ldots, f(y_{2n-1})) \right]_k .$$

Note that although this formulation includes y_j for $j > n$, there are really only $n+1$ distinct values being used since $y_{n-k} = y_{n+k}$. Also, $\Re$ denotes the real part of the Fourier transform and γ_k is defined as

$$\gamma_k = \begin{cases} 1 & k \in \{0, n\} \\ 2 & \text{otherwise.} \end{cases}$$

Problem 6. Write a function that accepts a function f and an integer n. Compute the $n+1$ Chebyshev coefficients for the degree n interpolating polynomial of f using the Fourier transform (`np.real()` and `np.fft.fft()` will be helpful). When using NumPy's `fft()` function, multiply every entry of the resulting array by the scaling factor $\frac{1}{2n}$ to match the derivation given above.

Validate your function with `np.polynomial.chebyshev.poly2cheb()`. The results should be exact for polynomials.

```
# Define f(x) = -3 + 2x^2 - x^3 + x^4 by its (ascending) coefficients.
>>> f = lambda x: -3 + 2*x**2 - x**3 + x**4
>>> pcoeffs = [-3, 0, 2, -1, 1]
>>> ccoeffs = np.polynomial.chebyshev.poly2cheb(pcoeffs)

# The following callable objects are equivalent to f().
>>> fpoly = np.polynomial.Polynomial(pcoeffs)
>>> fcheb = np.polynomial.Chebyshev(ccoeffs)
```

Lagrange vs. Chebyshev

As was previously stated, Barycentric Lagrange interpolation is $O(n^2)$ at startup and $O(n)$ at runtime while Chebyshev interpolation is $O(n \log n)$. This improved speed is one of the greatest advantages of Chebyshev interpolation. Chebyshev interpolation is also more accurate than Barycentric interpolation, even when using the same points. Despite these significant advantages in accuracy and temporal complexity, Barycentric Lagrange interpolation has one very important advantage over Chebyshev interpolation: Barycentric interpolation can be used on any set of interpolating points while Chebyshev is restricted to the Chebyshev nodes. In general, because of their better accuracy, the Chebyshev nodes are more desirable for interpolation, but there are situations when the Chebyshev nodes are not available or when specific points are needed in an interpolation. In these cases, Chebyshev interpolation is not possible and Barycentric Lagrange interpolation must be used.

Utah Air Quality

The Utah Department of Environmental Quality has air quality stations throughout the state of Utah that measure the concentration of particles found in the air. One particulate of particular interest is $PM_{2.5}$ which is a set of extremely fine particles known to cause tissue damage to the lungs. The file `airdata.npy` has the hourly concentration of $PM_{2.5}$ in micrograms per cubic meter for a particular measuring station in Salt Lake County for the year 2016. The given data presents a fairly smooth function which can be reasonably approximated by an interpolating polynomial. Although Chebyshev interpolation would be preferable (because of its superior speed and accuracy), it is not possible in this case because the data is not continous and the information at the Chebyshev nodes is not known. In order to get the best possible interpolation, it is still preferable to use points close to the Chebyshev extrema with Barycentric interpolation. The following code will take the $n+1$ Chebyshev extrema and find the closest match in the non-continuous data found in the variable `data` then calculate the barycentric weights.

```
>>> fx = lambda a, b, n: .5*(a+b + (b-a) * np.cos(np.arange(n+1) * np.pi / n))
```

```
>>> a, b = 0, 366 - 1/24
>>> domain = np.linspace(0, b, 8784)
>>> points = fx(a, b, n)
>>> temp = np.abs(points - domain.reshape(8784, 1))
>>> temp2 = np.argmin(temp, axis=0)

>>> poly = barycentric(domain[temp2], data[temp2])
```

Problem 7. Write a function that interpolates the given data along the whole interval at the closest approximations to the $n+1$ Chebyshev extremal nodes. The function should accept n, perform the Barycentric interpolation then plot the original data and the approximating polynomial on the same domain on two separate subplots. Your interpolating polynomial should give a fairly good approximation starting at around 50 points. Note that beyond about 200 points, the given code will break down since it will attempt to return multiple of the same points causing a divide by 0 error. If you did not perform the fix suggested in the `ACHTUNG` box, make sure not to pass in any points that exactly match the interpolating values.

Additional Material

The *Clenshaw Algorithm* is a fast algorithm commonly used to evaluate a polynomial given its representation in Chebyshev coefficients. This algorithm is based on the recursive relation between Chebyshev polynomials and is the algorithm used by NumPy's `polynomial.chebyshev` module.

Algorithm 9.1 Accepts an array x of points at which to evaluate the polynomial and an array $a = [a_0, a_1, \ldots, a_{n-1}]$ of Chebyshev coefficients.

```
1: procedure ClenshawRecursion(x, a)
2:     u_{n+1} ← 0
3:     u_n ← 0
4:     k ← n − 1
5:     while k ≥ 1 do
6:         u_k ← 2x u_{k+1} − u_{k+2} + a_k
7:         k ← k − 1
8:     return a_0 + x u_1 − u_2
```