# APC 523/MAE 507/AST 523
# Numerical Algorithms for Scientific Computing
# Problem Set 2
# (due Saturday ?? April 2019, 11:55am)

## 1 "Barycentric" interpolation formula

*(This problem is a lot shorter than it looks. 90+% of this verbiage is background info and context, and I explicitly walk you through an algorithm for this problem.)*

Given $N+1$ nodes $x_j$, $j = 0, 1, \ldots, N$ (all distinct, but not necessarily sequential) and a function $f$ with values $f_j \equiv f(x_j)$, $j = 0, 1, \ldots, N$ at those nodes, there is a unique polynomial interpolant $p_N(x)$ of degree $\leq N$ that agrees with $f$ exactly at those nodes. Associated with that set of nodes are $N+1$ degree-$N$ **Lagrange basis polynomials**

$$L_j(x) \equiv \prod_{\substack{k=0 \\ k \neq j}}^{N} \frac{x - x_k}{x_j - x_k} \tag{1}$$

such that the (unique) interpolating polynomial can be expressed as a linear combination of the $L_j$ (the known function values $f_j$ are the coefficients):

$$p_N(x) = f_0 L_0(x) + f_1 L_1(x) + \ldots + f_N L_N(x) = \sum_{j=0}^{N} f_j L_j(x) \quad . \tag{2}$$

This is the **Lagrange form** (or Lagrange representation) of $p_N$ that we saw in class.

Though not discussed in most textbooks, it turns out that equation (2) can be refactored into its so-called "barycentric" form[1]

$$p_N(x) = \begin{cases} \dfrac{\sum_{j=0}^{N} \frac{w_j^{(N)}}{x - x_j} f_j}{\sum_{j=0}^{N} \frac{w_j^{(N)}}{x - x_j}} & x \neq x_0, x_1, \ldots, x_N, \\ f_k & x = x_k \quad , \end{cases} \tag{3}$$

where the weights $w_0^{(N)}, w_1^{(N)}, \ldots, w_N^{(N)}$ will be defined below. In this problem, you will derive (3) and code up an efficient implementation of it (I'll walk you through the steps).[2]

The purpose of this problem is threefold:

- Many authors[3] contend that the Lagrange form, while nice for framing the theory of interpolation, is impractical for actually evaluating $p_N(x)$. Arguments include computational cost, inaccuracy at $x$ values near any of the nodes $x_j$, and the overhead incurred if you add new nodes later.[4] One point of this problem is to challenge this conventional wisdom, since the formulation in (3) sidesteps all of these shortcomings.

[1] It's called "barycentric" because of its resemblance to a center-of-mass formula, where the $f_j$ are "positions" and the $w_j^{(N)}/(x - x_j)$ are "masses".

[2] Our construction will largely follow the one in **Berrut & Trefethen (2004)**, with a few tweaks and clarifications. **I included a PDF copy along with this assignment.** This problem is self-contained and can be done without the article, but it's a well-written exposition of a lovely piece of applied mathematics that I recommend reading at some point.

[3] e.g. Epperson. See the last paragraph of Section 4.1 and the first paragraph of Section 4.2 for Epperson's assessment.

[4] Since the individual basis functions $L_j$ depend on the set of nodes, they must be recomputed from scratch (or at least heavily updated) any time that *set* changes.

- Equation (3) has some neat numerical properties, and some of what I ask you to do in the implementation will seem like bad practice (e.g. an `if` statement that tests for *exact* floating-point equality). So the barycentric interpolation formula is edifying in its own right.
- For **Chebyshev nodes of the second kind** (see Problem 2), the barycentric formula is faster and more accurate than almost any alternative, even when the interpolant has a degree $\sim 10^5$ or more. So you'll use the output from this problem as part of a general Chebyshev interpolator later.

So let's get to it…

**(a)** Based on equation (1), show that

$$\sum_{j=0}^{N} L_j(x) \equiv 1 \tag{4}$$

(we'll need this result later). *HINT: This shouldn't take any algebra. Just think about interpolating the constant function* $f(x) \equiv 1$.

**(b)** The $j$th Lagrange basis polynomial evaluates the difference between $x$ and every node *except* the $j$th node. Say you insert a factor of $(x - x_j)$ into both the numerator and denominator of (1). Show that $L_j(x)$ can then be rewritten as[5]

$$L_j(x) = \frac{w_j^{(N)}}{x - x_j} L^{(N)}(x) \quad , \tag{5}$$

where

$$L^{(N)}(x) \equiv (x - x_0)(x - x_1)\dots(x - x_N) = \prod_{k=0}^{N} (x - x_k) \tag{6}$$

and

$$w_j^{(N)} \equiv \prod_{\substack{k=0 \\ k \neq j}}^{N} \frac{1}{x_j - x_k} \quad . \tag{7}$$

Of course, expression (5) is only ok so long as $x \neq x_j$ (else it becomes indeterminate). So thinking about what the value of $L_j(x = x_j)$ should be, add a logical branch to (5) to cover the case $x = x_j$.

**(c)** By extension, show that the interpolating polynomial can be rewritten as

$$p_N(x) = \begin{cases} L^{(N)}(x) \sum_{j=0}^{N} \frac{w_j^{(N)}}{x - x_j} f_j & x \neq x_0, x_1, \dots, x_N \\ f_k & x = x_k \end{cases} \quad , \tag{8}$$

**(d)** Combine equations (4) and (5) to show that

$$L^{(N)}(x) \sum_{j=0}^{N} \frac{w_j^{(N)}}{x - x_j} \equiv 1 \quad . \tag{9}$$

Inject this as a denominator to the $x \neq x_j$ branch of (8), cancel the $L^{(N)}(x)$ and finally arrive at (3).

[5] The superscript $(N)$ in both cases denotes the highest index among the nodes in play, i.e. that these quantities are constructed from a set $x_0, x_1, \dots, x_N$ of $N + 1$ nodes.

**(e)** Now code up an efficient implementation of (3). Use any language you like (Python is probably simplest). You probably want to be object-oriented[6] and create a class `BaryInterp` (or some name like that) of callable objects that is initialized with a set of nodes $(x_j, f_j)$ and can then be evaluated when subsequently fed an argument $x$. This class should also have an `update()` method that allows you to add new nodes (assume we will never delete nodes). It will really slow you down if you have thousands of nodes.

There are a couple of major things to consider.

First, how do you compute the $w_j^{(N)}$ efficiently? Here's one way. Imagine a lower triangular array of quantities $w_j^{(K)}$, where $K$ indexes the row and $j$ the column (and since $w_j^{(K)}$ represents the $j$th weight when the node indices go up to $K$, this expression is undefined when $j > K$, ergo why the array I'm proposing is lower triangular). Define the upper left entry to be $w_0^{(0)} \equiv 1$. If we have $N+1$ nodes (node indices running from 0 to $N$), then row $K = N$ of this array gives us our weights.

Notice some properties of this array. If $j < K$, then the $(j, K)$ entry in the array is just the one above it times $1/(x_j - x_K)$:

$$w_j^{(K)} = w_j^{(K-1)} \left( \frac{1}{x_j - x_K} \right) \quad . \tag{10}$$

Furthermore, each diagonal entry $w_K^{(K)}$ is just the product of the factors $1/(x_j - x_K)$ with $j < K$ times $(-1)^K$.

Together, these facts suggest a procedure like the following. Say I give you nodes $x_0, \ldots, x_N$.

(i) Fill in all the sub-diagonal entries of our conceptual lower-triangular array with the pairwise differences $x_j - x_K$ between nodes.

(ii) Make each diagonal entry $w_K^{(K)}$ the product of all the entries to its left times $(-1)^K$ ($w_0^{(0)}$ must be manually initialized to 1).

(iii) Multiply every entry in the bottom row $N$ by all the entries directly above it in the same column (i.e. all the entries with the same $j$).

(iv) Take the reciprocal of each entry in row $N$.

Voila — the last row should now be the needed weights $w_j^{(N)}$, and you can discard the rest of the table. You don't actually need to create a whole lower triangular array to execute the steps above — some nested `for` loops will also do the job. Also, I worked with products as intermediate quantities and only flipped them over at the end just to avoid unnecessary division operations en route.

You don't have to generate the weights this way, particularly if you find a more efficient scheme. It's just a suggestion meant to save you thinking time.

Updating is also now fairly straightforward. Suppose you currently have the $N+1$ weights $w_j^{(N)}$ and want to add, say, 5 new nodes. We imagine that the lower triangular array is back and that we are going to extend it by 5 rows. Just follow the procedure above but for only the lower "trapezoidal" array that starts at row $N+1$, and after the last step, multiply each final entry in row $N+5$ by any entries in row $N$ that would be found "above" an entry in row $N+5$ (you could just extend row $N$ by appending five 1s to the end and then multiply the two arrays, leveraging vectorized arithmetic).

The second major consideration is how to handle $x$ values close to one of the nodes. If $x$ is too close to $x_k$, you should probably just return $f_k$, but how close is too close? Won't we suffer major loss of significance once we start subtracting two nearly equal quantities and, worse yet, dividing by

[6] If you aren't comfortable writing classes and want to implement this as a function that takes $x$ and the nodes as arguments and re-evaluates everything from scratch each time, that's ok, but I discourage it.

that tiny value to create a dominant quantity in each summation that has lost significance? Figuring out how to guard against all these contigencies seems like a lot of trouble, and you might fear that you'll still be left with very inaccurate answers when $x \approx x_k$.

Interestingly, you should be totally ok by only returning $f_k$ when $x$ is *exactly* equal to $x_k$. You should get accurate answers from (3) even if $x$ barely differs from one of the $x_k$. That may shock you, and it's worth understanding what's going on. Yes, the subtraction destroys significance — the $k$th term in each of the sums in the numerator and denominator of (3) becomes quite wrong. However, the wronger it gets, the more dominant it becomes in the respective sums, so that the ratio of the top to the bottom nevertheless approaches $f_k$. This is the main reason to "divide (8) by 1" to generate (3). When you divide two quantities whose individual wrongnesses are being magnified by a similar/identical significance-crushing subtraction, the contributions to the net relative error from the numerator and denominator will tend to have opposite signs and exhibit considerable cancellation between them, leaving you with a very small error. Try it.

**(f)** Test your code by evaluating it on

$$f(x) = x^4 + 2x^3 + 3x^2 + 4x + 5 \tag{11}$$

over the interval $-1 \leq x \leq 1$. This is an actual polynomial, so we should be able to reproduce it. For the "exact" values, use a built-in polynomial class from Numpy or MATLAB or whatever (just so it's evaluated intelligently). Use three different sets of 5 nodes each:

- Equally spaced nodes (including $-1$ and 1);[7]
- The 5 roots of the Chebyshev polynomial $T_5(x)$;
- The 5 maxima/minima of the Chebyshev polynomial $T_4(x)$.

In each case, plot the relative error and report that maximum relative error on $[-1, 1]$.

**Epilogue** Since they don't usually get into the barycentric form (at least not in detail), many textbook authors eschew Lagrange interpolation in favor of interpolation based on the **Newton form**

$$p_N(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \ldots + c_N(x - x_0)(x - x_1)\cdots(x - x_{N-1}) \quad . \tag{12}$$

In most implementations of Newton interpolation, the coefficients $c_j$ are computed via **divided differences**[8], and the actual evaluation at $x$ is accomplished either via **Horner's algorithm**[9] or **Neville's algorithm**[10]. I leave it to you to read about Newton interpolation, divided differences, etc. Sometimes, it does have advantages over barycentric Lagrange interpolation (the Berrut & Trefethen paper has a nice comprehensive discussion). But it's not like Newton interpolation is amazing and Lagrange interpolation is useless, as is often stated. Far from it, if you use the barycentric form.

Just for completeness, one thing that you should pretty much never do is use the **monomial form** of $p_N$

$$p_N(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_N x^N \quad . \tag{13}$$

If you have the coefficients $a_j$, you should evaluate $p_N(x)$ using a variant of Horner's algorithm (see, e.g. Epperson Section 2.1). However, determining the $a_j$ from the $x_j$ and $f_j$ in the first place is usually an ill-conditioned problem.[11] So this entire line of attack is bad news. For evaluting polynomial interpolants, stick to Newton or (barycentric) Lagrange.

[7] Since the function we're interpolating is an actual polynomial, using equally-spaced nodes shouldn't pose a problem.

[8] See, e.g., Section 4.2 of Epperson.

[9] Essentially nested multiplication. See Algorithm 4.2 on pg. 179 of Epperson, which implements a nested multiplication that he outlines (in a different context) in Section 2.1.

[10] A recurrence relation to evaluate $p_N$ in terms of intermediatley computed lower degree polynomials. See NR Section 3.2.

[11] Look up "Vandermonde matrix" and see discussion in *Numerical Recipes* Sections 3.5 and 2.8.1.

## 2 Chebyshev nodes of the second kind

*(This problem is also a lot shorter than it looks. A few lines of code to directly implement two formulas from the Berrut & Trefethen paper, and then making a bunch of plots for which I give explicit instructions. As before, most of the verbiage is background info and context.)*

Consider a function $f$ defined on $-1 \leq x \leq 1$[12] that you want to interpolate through $f$'s values at $N+1$ nodes. Different choices of the $N+1$ nodes yield different interpolating polynomials, but each choice yields a unique[13] interpolant of degree $\leq N$. We asserted in class that if the nodes are the $N+1$ roots

$$x_j \equiv \cos\left(\frac{2j+1}{N+1}\frac{\pi}{2}\right), \quad j = 0, 1, \ldots, N \tag{14}$$

of $T_{N+1}(x)$,[14] then the $\infty$-norm error $||f - p_N||_\infty$ between $f$ and the resulting interpolant $p_N$ will be minmized. These points are known as **Chebyshev points of the first kind** (or **Chebyshev-Gauss points**).

But there is another set of similarly distributed $N+1$ nodes that can be more useful in certain contexts, namely the *extrema*[15]

$$x_j \equiv \cos\left(\frac{j}{N}\pi\right), \quad j = 0, 1, \ldots, N \tag{15}$$

of $T_N$. While technically not as optimal as the points in (14), these **Chebyshev points of the second kind** (or **Chebyshev-Lobatto points**) are pretty close to optimal for practical purposes (meaning that maybe you need a few more nodes to get the same accuracy). But they have other compelling advantages:

- the endpoints -1 and 1 are always in node-set (15) (but never in node-set (14)). This matters because, very near the ends of your interval, you are always technically *extrapolating* rather than *interpolating* when you use set (14) (since you're going outside the span of the nodes), and that introduces a tiny bit of error near the edges. For boundary value problems, in which accuracy near the boundaries is important, this can make a difference.

- the weights $w_j^{(N)}$ for both node-sets (14) and (15) have simple analytic forms[16], but while the weights for (14) still require a little bit of computation to evaluate, the weights for (15) *require no computation at all!*

  To get a sense of why this matters, say we're given $N+1$ generic nodes. Evaluating $p_N(x)$ only takes $\mathcal{O}(N)$ operations *once you have the weights*, but it takes $\mathcal{O}(N^2)$ operations to get the weights in the first place. For nodes in (14) , the weights can be computed with $\mathcal{O}(N)$ evaluations of a trig function. If you change $N$ to get a new set of nodes, you have to recompute the weights for the new node-set, incurring an $\mathcal{O}(N)$ overhead. In contrast, the weights for nodes in set (15) are *constants* that can be adjusted for a new $N$ in essentially zero time with essentially zero overhead.

  These facts make the Chebyshev-Lobatto nodes togther with the barycentric formula an *extremely* potent one-two punch for constructing and then evaluating polynomial interpolants even with very large $N$.

[12] If our interval is instead $[a, b]$, we can always map it to $[-1, 1]$ by a linear change of variable.

[13] We may represent that interpolant in a variety of forms — monomial form, Lagrange form, Newton form, or as a sum of Chebyshev polynomials $T_0(x), T_1(x), \ldots, T_N(x)$ of degree $\leq N$ — but those are just different representations of what is a single underlying polynomial.

[14] The highest degree Chebyshev polynomial that would be part of a degree-$\leq N$ interpolant is $T_N$, so these are roots of the *next*-degree Chebyshev polynomial (which, of course, wouldn't appear in a Chebyshev representation of $p_N$).

[15] $T_N$ is a degree-$N$ polynomial with $N$ distinct roots on $[-1, 1]$ and (when $N \geq 1$) $N+1$ distinct extrema ($N-1$ points where the derivative is zero plus the two endpoints of the interval, at which $T_N$ always takes on extreme values of $\pm 1$).

[16] The weights also have analytic forms for equispaced nodes. See Section 5 of Berrut & Trefethen (2004).

In this problem, you will modify your barycentric interpolator from Problem 1 to accept optional precomputed weights (to remove that $\mathcal{O}(N^2)$ overhead) and then explore some interpolation phenomena with that tool.

**(a) Slightly modify the barycentric interpolator you build to accept pre-computed weights** Add an optional argument (in Python, this means a keyword argument or "kwarg") to the `__init__()` and `update()` methods so that the user can sidestep the computation of weights by providing them as an array.[17]

**(b) Write short functions to pre-compute the weights for Chebyshev nodes** Equations (5.3) and (5.4) in Berrut & Trefethen (2004) give the weights for Chebyshev points of the first and second kind, respectively:

$$w_j^{(N)} = (-1)^j \sin\left(\frac{2j+1}{N+1}\frac{\pi}{2}\right) \qquad \text{1st kind} \tag{16}$$

$$w_j^{(N)} = \begin{cases} \frac{(-1)^j}{2} & j = 0 \text{ or } j = N \\ (-1)^j & 1 \le j \le N-1 \end{cases} \qquad \text{2nd kind} \quad . \tag{17}$$

I believe the article does the (not too complicated) derivations if you're interested.[18]

Write two standalone functions `chebweights1()` and `chebweights2()` to generate the appropriate weights. Unless you want to recompute the nodes each time you call it, `chebweights1()` should probably just take an array of Chebyshev nodes of the first kind as its input. `chebweights2()`, on the other hand, can just take an integer argument.[19]

Finally, make sure that, given some function $f$, you can quickly generate a barycentric interpolator for $f$ using any number of Chebyshev nodes of either the first or second kind (we'll use interpolators with varying $N$ value in the next few parts of the problem). The steps would be (i) make an array of $N+1$ Chebyshev nodes (of either type), (ii) generate the corresponding weights (the weights are independent of $f$), (iii) evaluate $f(x)$ at those nodes to get the $f_j$, and (iv) feed all this into your barycentric interpolator constructor to get a callable object. There's nothing to submit here. Just make sure you can produce a functioning Chebyshev interpolant as needed, since we'll need them for parts (c) and (d).

To help cut down on work, in Numpy version 1.16 (not sure about earlier versions), there are some utility functions that crank out Chebyshev nodes (of both kinds) for you (but read this[20] footnote). They are buried in `numpy.polynomial.chebyshev` and are called `chebpts1()` and `chebpts2()`, respectively (each takes a single argument: the number nodes you want, i.e. what we've been calling $N+1$).

**(c) The Runge phenomenon**

[17] Assuming you're doing this in Python, then in order not to break any code that relies on the code as written in Problem 1, you should probably give the `weights` kwarg a default value of `None` and only compute the `wj` when `weights is None`.

[18] Technically, the weights have additional factors that are independent of $j$, but the nice thing about the barycentric formula (3) is that any such factors cancel in the top and bottom. We added that denominator-which-is-really-1 with foresight, precisely so that we could leverage this cancellation.

[19] Should this argument denote the degree of the interpolating polynomial, or the number of nodes being used (which is one larger)? To keep with our convention so far of calling the number of nodes $N+1$, I'd probably go with the former option and call the argument $N$. To me that seems "cleanest", but you're welcome to make an alternate choice.

[20] Small rounding errors are introduced by using (14) and (15) as written to generate the nodes, which the Numpy utility functions do. When we get to computing actual Chebyshev coefficients (see Problem 3), this has the annoying effect of making some coefficients that should vanish by symmetry instead come out to values of order $10^{-17} - 10^{-16}$. There is an easy and neat trick to remove this spurious rounding error altogether via a simple re-write of (14) and (15). This trick is also applicable to Fourier coefficients, and it offers some insight into how cos and sin are implemented under the hood in the source files for the C math library `math.h`. If enough of you are curious about it, let me know and I'll write it up as optional problem.

(i) Make three successive polynomial interpolants to $e^x$ using 4, 8, and then 16 equally-spaced nodes on $[-1, 1]$ (include both endpoints of the interval among your nodes). Evaluate these things at 1001 abscissae (again, including the endpoints) to get good domain coverage, plot the relative error for all three interpolants, and report the maximum (relative) error for each interpolant. Spoiler alert: $e^x$ should turn out to be interpolated just fine with equispaced nodes.

(ii) Repeat the same exercise for the function[21]

$$g(x) = \frac{1}{1 + 25x^2}, \quad -1 \leq x \leq 1 \quad . \tag{18}$$

This time, things should get *worse* as the number of nodes goes up.[22]

If you're wondering why this function is badly interpolated but the exponential function is ok, it's related to how close $g(x)$'s poles in the complex plane are to the segment $[-1, 1]$ of the real axis.[23] $e^x$ is, of course, entire (analytic in the whole complex plane), so it had no poles to worry about.

(iii) Now interpolate $e^x$ using 4, 8, and then 16 Chebyshev nodes of the first kind (which don't include the endpoints) and of the second kind (which do include the endpoints). Again, plot the relative error over $[-1, 1]$ and report the maximum relative error. Comment on salient differences you notice (if there are any) between the two kinds of interpolants in terms of maximum error and in terms of how the error is distributed over different parts of the interval $[-1, 1]$.

(iv) Repeat the previous part for the function (18). Spolier alert: Chebyshev nodes interpolate $g(x)$ just fine.

**(d) Convergence rate of Chebyshev interpolants** For this part of the problem, we're going to compute a lot of interpolants with varying numbers of points, so let's use only Chebyshev nodes of the second kind (since there's no expense in getting the weights for the barycentric formula).

Imagine interpolating a function $f$ with successively more Chebyshev nodes (15), i.e. with polynomial interpolants $p_N$ of successively higher degree $N$. Recently in class, I mentioned in passing that this sequence of interpolants will converge to $f$ as $N \to \infty$ if $f$ is **Lipschitz continuous** on $[-1, 1]$.[24] But how *quickly* do the interpolants converge as a function of $N$? In other words, as I add nodes, how quickly does $||f - p_N||_\infty$ drop?

It depends on how smooth $f$ is. I'll assert the answer for different degrees of smoothness without proof and ask you to make a few plots to validate these assertions empirically.

- Suppose all derivatives of $f$ up to the $\kappa$th derivative have a finite $L^1$-norm[25]

$$\left\| f^{(\kappa)} \right\|_1 \equiv \int_{-1}^{1} dx \left| f^{(\kappa)}(x) \right| \tag{19}$$

on $[-1, 1]$. Said another way, suppose that the $(\kappa - 1)$th derivative is continous. Then the Chebyshev interpolants of $f$ will converge at a rate that goes like $\mathcal{O}(1/N^\kappa)$. So on a log-log plot of $||f - p_N||_\infty$ vs. $N$, we should see roughly a line of slope $-\kappa$.

[21] This is the classic example of a function that's poorly interpolated by a single polynomial using equispaced nodes.

[22] Although you don't have to submit it, it's instructive to plot the interpolants overlaid on $g(x)$ just for yourself, to visually drive home the badness of the interpolant near the edges.

[23] See, e.g., Demanet and Ying (2010). It's a fairly math-y paper, but the first couple of pages might give you some idea of terms to look up for more information. For example, "Bernstein ellipses".

[24] In a nutshell, this means that any secant line you would draw between any two points on $f$ has an upper bound on the magnitude of its slope.

[25] When $f^{(\kappa)}$ has this property, we say that the $\kappa$th derivative of $f$ has a **bounded total variation**. In practice, it means that $f^{(\kappa)}(x)$ is allowed to have a finite number of finite jumps. For instance, $|x|$ has a first derivative of bounded variation

- Suppose $f$ is infinitely continuously differentiable (i.e. $C^\infty$) on $[-1,1]$]. Then the convergence is exponential (geometric), going like $\mathcal{O}(1/\tau^N)$ for some constant $\tau$. So on a semilog plot (log scale only on the vertical axis) of $||f-p_N||_\infty$ vs. $N$, we should see roughly a line of negative slope.[26]

- Suppose $f$ is **entire** (analytic everywhere in $\mathbb{C}$). Then the convergence is faster than geometric, sometimes referred to as **spectral convergence** or **spectral accuracy**. So on a semilog plot (log scale only on the vertical axis) of $||f-p_N||_\infty$ vs. $N$, we should see a curve that slopes downward away from a line until possibly it levels out due to rounding error.

Let's make plots using a few test functions to see all of this in action...

(i) Each function below is differentiable $\kappa$ times on $[-1,1]$ (possibly with jumps in the $\kappa$th derivative), for different values of $\kappa$:[27]

$$f(x) = |x| \ , \qquad \kappa = 1 \tag{20}$$

$$f(x) = |\sin(5\pi x)|^3 \ , \quad \kappa = 3 \tag{21}$$

$$f(x) = |\sin(3\pi x)|^5 \ , \quad \kappa = 5 \quad . \tag{22}$$

For each function, generate polynomial interpolants with $N+1 = 1, 2, 4, \ldots, 256$ nodes. Evaluate $f$ and each interpolant at 10,001 equally spaced abscissae (including both endpoints) and determine the maximum absolute error $||f-p_N||_\infty$ for each $N$.[28] Plot of $||f-p_N||_\infty$ vs. $N$ on a log-log scale (just plot the ($N$,error) pairs as dots; don't join them with lines). For comparison, also plot $1/N^\kappa$ (which should be a line on the log-log scale). See whether this supports the claim above about the convergence rate of the interpolants with increasing $N$.

(ii) Each function below is $C^\infty$ on $[-1,1]$ but has points of non-differentiability elsewhere:

$$f(x) = \frac{1}{1+25x^2} \tag{23}$$

$$f(x) = \sqrt{x+3} \tag{24}$$

$$f(x) = \tan x \quad . \tag{25}$$

For each function, generate polynomial interpolants with $N+1 = 1, 2, 4, \ldots, 256$ nodes. Evaluate $f$ and each interpolant at 10,001 equally spaced abscissae (including both endpoints) and determine the maximum absolute error $||f-p_N||_\infty$ for each $N$. Plot $||f-p_N||_\infty$ vs. $N$ on a semilog scale (just dots). If the claim above about the convergence rate of the interpolants to $C^\infty$ functions is correct, you should see something roughly linear.

(iii) Each function below is entire:

$$f(x) = \cosh x \tag{26}$$

$$f(x) = e^{-x^2} \tag{27}$$

$$f(x) = \cos(100\pi x) \quad . \tag{28}$$

For each function, generate polynomial interpolants with $N+1 = 1, 2, 4, \ldots, 256$ nodes. Evaluate $f$ and each interpolant at 10,001 equally spaced abscissae (including both endpoints)

[26] The slope would be $-\log\tau$, but we don't have any way to know what $\tau$ is, so all we can say is that the semilog plot should look linear with *some* slope.

[27] FYI, the chain rule applied to absolute value implies that $d\,|u(x)|\,/dx = u'(|u|\,/u)$ (you can check this by rewriting $|u|$ as $(u^2)^{1/2}$ and differentiating normally. Differentiate $y = |x|^3$ a few times and you should see that $y'$ and $y''$ are continuous, while $y'''$ has a finite jump discontinuity.

[28] This amounts to computing the array $\texttt{abs}(f(x) - p_N(x))$ and getting the maximum entry in that array.

and determine the maximum absolute error $||f - p_N||_\infty$ for each $N$. Plot $||f - p_N||_\infty$ vs. $N$ on a semilog scale (just dots). If the claim above about the convergence rate of the interpolants to entire functions is correct, you should see something that slopes down more steeply as $N$ increases.

# 3 Chebyshev coefficients via the DCT

*(From this point forward, unless otherwise specified, when I say "Chebyshev nodes" or "Chebyshev points", I mean the ones of the* ***second kind*** *listed in (15)).*

*And as with all these problems, it's much shorter than it looks. The verbiage is to guide you through things and fill in gaps.*

An interpolating polynomial $p_N$ can be represented in variety of forms: monomial form, Newton form, Lagrange form, barycentric Lagrange form, etc. It can also be represented as a (finite) linear combination

$$p_N(x) = \sum_{j=0}^{N} c_j T_j(x) \tag{29}$$

of Chebyshev polynomials, whether or not the nodes being used are Chebyshev nodes. Up to now, we haven't made use of the Chebyshev *coefficients* of an interpolating polynomial $p_N$ because we haven't needed them. For the purpose of evaluating $p_N(x)$ at arbitrary $x$, the barycentric form does just fine.

But the $c_j$ are useful for other purposes (among them, differentiation and integration), so in this problem, we'll see how to compute them. For a generic set of $N+1$ nodes, finding the $c_j$ can be as problematic as finding the monomial coefficients is: you solve a linear system that has a Vandermonde-like matrix, a procedure that isn't always stable or accurate. But if you have the $f_j$ at Chebyshev nodes, you can leverage the FFT (more specifically, an FFT-based implementation of the discrete cosine transform, or DCT) to get the $c_j$ quickly and accurately. There are some pitfalls, so this problem walks you through the explicit steps for doing this for interpolants through Chebyshev nodes of the second kind (I leave it to you to look up the analogous construction using Chebyshev nodes of the first kind).

**Aside on Numpy's `chebinterpolate()` function**

Given a function $f$ and a degree $N$, the routine `chebinterpolate()` in `numpy.polynomial.chebyshev` returns Chebyshev coefficients for the degree-$N$ polynomial interpolant to $f$. Why not just use that?

First, `chebinterpolate()` can only return the interpolant through Chebyshev points of the first kind. You'd think this routine would have an optional argument to switch to points of the second kind. It doesn't. So it's limiting.

Second, that routine uses the Vandermonde method under the hood, and on top of that, its implementation (which I haven't perused in detail) seems shortsighted, in that it breaks if you ask it for too many coefficients.[29] In contrast, the DCT-method below returns $\sim 1$ million coefficients almost immediately with no issues.

Moral of the story — don't just use library routines without having some idea how they do what they do. Sometimes they don't quite have the specific functionality you want, or they're implemented in ways that aren't robust enough for your use case, or both. If you do need to roll your own routine, it's still ideal to leverage other library routines within your custom routines when possible. For instance, in this problem, we'll use

[29] Try asking for the coefficients up to $N = 10^6$ for a nice analytic function like $\cosh x$. It throws a memory error.

Scipy's DCT routines, which in turn use a well-developed Fortran FFT library under the hood.

The actual work for this problem will be: write a function that returns the $N+1$ $c_j$ coefficients in (29) given the $f_j$ values at $N+1$ Chebyshev nodes (15); run a couple of validation tests to make sure you did it right; and then run that function for a few different $N$ values so that you can see what makes a good vs. bad choice of $N$. I'll lay all this out below in parts (a), (b), etc for this problem, but first, some background so that you understand which variant of the DCT we will use (there are several) and why.

**A Chebyshev series as a Fourier (cosine) series**

If I hand you a generic function of $\phi$ that is both *periodic* on $[0, 2\pi]$ and *even* ($f(-\phi) = f(\phi)$), then its Fourier series will have only cosine terms, the first $N+1$ of which will look like

$$f(\phi) = \frac{a_0}{2} + \sum_{j=1}^{N} a_j \cos(j\phi) \tag{30}$$

(the convention for Fourier series is to have that factor of $1/2$ in the constant term). Recall that Chebyshev polynomials satisfy

$$T_k(\cos\phi) \equiv \cos(k\phi) \quad . \tag{31}$$

The nodes (15) all have the form $\cos\phi$ for some $\phi$, so the Chebyshev expansion (29) is equivalent to

$$p_N(x = \cos\phi) = \sum_{j=0}^{N} c_j \cos(j\phi) \quad . \tag{32}$$

The series (32) looks a lot like the Fourier cosine series (30), with three salient differences:

(i) the series (32) is technically only defined from 0 to $\pi$ (it was really a function of $x$ running from $-1$ to 1, and $\phi$ is an arccosine of $x$, so $\phi$ runs from $\pi$ to 0). Wait, $\pi$ to 0?! That seems backward, and is a good segue into…

(ii) when the underlying $x$ values in (32) run in increasing order, the corresponding $\phi$ values run in *decreasing* order. And finally…

(iii) there's no factor of $1/2$ in the $c_0$ term of (32) (for no reason other than that the conventions differ).

Let's address these issues in turn…

(i) There's a natural way to extend (32) into $[\pi, 2\pi]$. Notice that in addition to being even (having reflection symmetry around $\phi = 0$), every function $\cos(j\phi)$ also has reflection symmetry around $\phi = \pi$:

$$\cos\left(j(\pi + \delta)\right) = \cos\left(j(\pi - \delta)\right) \quad . \tag{33}$$

So we can extend our function naturally by assigning it the same $y$-values at points a distance $\delta$ to the right of $\phi = \pi$ as we have at points a distance $\delta$ to the left of $\phi = \pi$. In that sense, the $N+1$ discrete $f_j$ values on $[0, \pi]$ are really encoding discrete values all the way out to $\phi = 2\pi$. Most off-the-shelf DCT routines[30] take this extension (plus the even-ness of the function) into account under the hood in some optimized way as they execute a modified FFT transformation that converts real-valued $f_j$'s into (also real-valued) $a_j$'s without needlessly computing fully complex

[30] I believe the one in `scipy.fftpack` employs the method described in this paper.

coefficients (the way the FFT typically does) or needlessly computing terms that will be zero given the evenness of the function.

But there's still some ambiguity. For instance, if I hand you a discrete set of $f_j$ values at $N+1$ Chebyshev nodes of the *first* kind, then $f_0$ corresponds to a point just to the right of $\phi = 0$ and $f_N$ to a point just to the left of $\phi = \pi$, so all $N+1$ values need to be "reflectively cloned" into $[\pi, 2\pi]$ for a total of $2(N+1)$ values. On the other hand, using nodes of the *second* kind (as we're doing here), $f_0$ and $f_N$ don't need to be cloned.[31]

A DCT routine doesn't know what nodes you chose. It just receives an array of $N+1$ $f_j$ values that it takes to be equispaced in $\phi$, but beyond that, we need to tell the routine what the reflection properties at the left and right ends of the $[0, \pi]$ interval are. In general, there are 8 different cases[32], so there are 8 types of DCT implementations (the "Informal overview" section of the Wikipedia entry on "Discrete cosine transform" lays this out nicely with some pictures). The one we want for Chebyshev nodes of the second kind is called the **type-I DCT**.

So if we're using Python, we'll need to feed our $f_j$ into `scipy.fftpack.dct()` with the keyword argument "`type=1`".

(ii) Actually, we need to feed in the $f_j$ *in reverse*. Numpy arrays support a 3-entry colon-separated index-slicing notation that looks like `arr[start:end:step]`. If you omit "start" and "end, it just gives you back the whole array. If you omit "step", it steps by 1, as usual. If you let "step" be -1, it gives you a view of the array in reverse order. So the easiest way to accomplish this is to pass `fj[::-1]` into the DCT routine. If you fail to do this, all your odd-index coefficients will end up with the wrong sign.

(iii) Divide both the 0th and $N$th coefficients by 2. Why the last? It has to do with the fact that $f_N$ wasn't treated as being reflect-cloned into $[\pi, 2\pi]$, it gets half the value it should. But I don't want to digress on that here — read about the details of the DCT to learn more.

(iv) Normalization — this implementation of a type-I DCT only returns unnormalized coefficients. We now have to divide the whole coefficient array by $N$ (not $N+1$, just $N$ — one less than the number of nodes).

At the end of this process, you should have the Chebyshev coefficients of the polynomial that interpolates your function $f$ through $N+1$ Chebyshev nodes of the second kind.

Now let's put this into action…

**(a)** Write a routine `chebcoeffs()` that returns the $N+1$ Chebyshev coefficients for the interpolant $p_N$ of $f$ through $N+1$ Chebyshev nodes (2nd kind). The input should be an array of $f_j$ function values, assumed to be at Chebyshev nodes.

**(b)** To make sure it works, let's test it on the function

$$f(x) = 8x^4 + 4x^3 + 2x^2 + x + 1 \quad . \tag{34}$$

[31] We don't need to reflect-clone $f_N$ because it's right on $\phi = \pi$. We don't need to reflect-clone $f_0$ because it's right on $\phi = 0$, which would reflect to $\phi = 2\pi$. But discrete Fourier transforms (of which the FFT is an implementation) don't need you to supply the value at $2\pi$ because it's already assumed to be the same as the value at 0 (the assumption of $2\pi$-periodicity is "built into" the algorithm).

[32] Each end of the array is either a value right at the interval endpoint or just inside the endpoint (2 options), and the reflection across the endpoint could either be a straight reflection (which makes the extension continuous) or an anti-reflection (which makes it discontinous) (2 more options), and each end of the array can be treated separately in this regard (another factor of 2 as we count options, bringing us to 8). The pictures in Wikipedia, along with the itemization of the 8 types, helps to clarify this.

(i) Re-express $f(x)$ in a Chebyshev basis so you know what coefficients to expect. *Hint: It is useful first to express each of the monomials $1, x, x^2, \ldots$ in terms of Chebyshev polynomials. For instance, it's pretty clear that some linear combination of $T_0, T_2$, and $T_4$ can leave you with just $x^4$, and you can find that combination by inspection.*

(ii) Confirm that your `chebcoeffs()` returns the correct coefficients if you evaluate $f$ at 5 Chebyshev nodes (2nd kind).

**(c)** Now let's take another function, say

$$f(x) = e^{-x^2} \quad . \tag{35}$$

Get the Chebyshev coefficients of the interpolating polynomial $p_N$ to this function through $N+1$ Chebyshev nodes (2nd kind) when

- $N + 1 = 2^{10}$ (i.e. degree $2^{10} - 1$, so $2^{10}$ nodes)
- $N + 1 = 2^{13}$ (i.e. degree $2^{13} - 1$, so $2^{13}$ nodes)
- $N + 1 = 2^{13} + 1$ (i.e. degree $2^{13}$, so $2^{13} + 1$ nodes)

You don't need to even look at the coefficients, and you don't need to formally time these things (yet – you will below). Just casually observe. You should see that the first run is pretty quick, the second has a slight but noticeable "hiccup", and the third is also quick, even though it has one more node than the second run. To drive this home, now try the following numbers of coefficients:

- $N + 1 = 2^{17}$ (i.e. degree $2^{17} - 1$, so $2^{17}$ nodes)
- $N + 1 = 2^{17} + 1$ (i.e. degree $2^{17}$, so $2^{17} + 1$ nodes)

The former, which has one less node, might take up to a minute, during which the CPU fan on your laptop might even turn on. The latter should be pretty much immediate. What's going on here?

> **Know how your FFT implementation feels about array lengths**
>
> There is no single FFT algorithm; there are several, and most operate under the premise the matrix operations which underlie converting from physical space to Fourier space can be factored into a bunch of operations with smaller matrices that are faster to execute. Many implementations rely on the array of $f_j$ values being operated on to have a length that is factorable into powers of small-ish primes. If the library routine you've chosen works like this under the hood, and you try to do an FFT on an array with a number of entries equal to *large prime*, the run time will generally scale very poorly, like $\mathcal{O}(N^2)$.
>
> What about the DCT? Well, it depends which type of DCT you're doing, i.e. whether the end-values of the input array themselves get "reflect-cloned" into $[\pi, 2\pi]$ or whether the reflection happens *right at* those nodes. In our case of type-I DCT, when we imagine "folding" the function over $\phi = \pi$ so that it spans the whole interval $[0, 2\pi]$, $f_0$ and $f_N$ are *not* reflected. So in total, our DCT is kind of like doing an FFT with $N+1$ points from 0 to $\pi$ (inclusive) and then another $N - 1$ cloned points from just past $\pi$ all the way to $2\pi$. That's $2N$ total points on $[0, 2\pi]$, and in Scipy's implementation, things work fastest when that total number $2N$ of points is highly factorable. It works fastest when $2N$ is a power of 2, which means when $N$ itself is a power of 2. So that means it will work fastest when our original number $N + 1$ of nodes on $[0, \pi]$ is one more than a power of 2.
>
> If we make the number of nodes $2^{13}$, then the degree $N$ of the interolating polynomial is $2^{13} - 1 = 8191$, which is a Mersenne prime. Ditto for $2^{17} - 1 = 131071$. With a number

of equivalent-FFT-points that equals a power of 2, Scipy's DCT implementation gets $\mathcal{O}(N \log_2 N)$ time scaling that the FFT is known for. But with a number of equivalent-FFT-points, it's more like $\mathcal{O}(N^2)$.

At this juncture, using Python's `timeit.timeit()` function from the `timeit` module to actually clock these DCTs is instructive.

**Write a short script (or some command-line lines) to time the calculation of the Chebyshev coefficients for $8, 2^{13}, 2^{13}+1, 2^{17}$, and $2^{17}+1$ $f_j$ values for the function (35) above.** Submit your code (shouldn't be long) and the result of the running times. **WARNING!** `timeit()` defaults to trying $10^6$ runs of whatever statement you give it, so pass a more sensible value like 10 with the keyword argument `number=10` (that should make your longest run take 3-5 minutes total). If you're writing in a different language that has a simple timing/profiling tool that you're more comfortable with, you're welcome to use that.

**Then measure the run time for calculating coefficients using $N+1 \equiv 2^k+1$ $f_j$ values, for $k = 10, 12, 14, \ldots, 24$. Plot run time vs. $N$ on a log-log plot and confirm that the scaling is somewhere between $\mathcal{O}(N)$ and $\mathcal{O}(N^2)$.** Do NOT try compute the DCT with something like $N+1 = 2^{23}$ nodes! It will take forever. If the number of nodes is *one more* than a power of 2, you're cool. If $N+1$ is exactly a power of 2, and you get unlucky enough that $N$ itself is prime or barely factors, your CPU will be unhappy.

Just crudely on my laptop, jumping from 8 nodes to $2^{13}$ nodes (which gives us $N = 2^{13}-1$, which is prime and bad) increases the computational time by a factor of $\sim 150$. Going from 8 to $2^{13}+1$ nodes (i.e. $N = 2^{13}$) instead increases the cost by a factor of only $\sim 5$. The absolute runtimes are still fractions of a second, so you barely notice any difference, but it's there. Likewise, going from $2^{13}$ to $2^{17}$ nodes increases run time by another factor of $\sim 250$, bringing the absolute run time up to 18 seconds or so, which is definitely noticeable. Going instead from $2^{13}+1$ to $2^{17}+1$ nodes increases run time by a factor of only $\sim 10$ and keeps the absolute compute time (for one run of the coefficients, not 10 runs) to around 25 milliseconds.

**Moral(s) of the story**

1. Chebyshev coefficients can be computed via an FFT-like process called the DCT. This works because a Chebyshev representation can be viewed as a condensed Fourier representation of a real-valued function with certain symmetries.
2. The DCT is usally very fast, but it's important to be aware enough of your implementation to know what lengths of arrays it handles best. With "vanilla" FFT implementations and their variants, it's hard to go wrong with array lengths that are powers of 2. But inauspicious choices can sabotage your computational savings.

**Epilogue: Clenshaw's recurrence formula** Having the $c_j$ opens up an alternate way to evaluate $p_N(x)$ called **Clenshaw's recurrence relation**, discussed in Sections 5.4, 5.4.1, and 5.4.2 of *Numerical Recipes*. Like barycentric interpolation, it is fast, accurate, and stable. I don't happen to know which is faster, but if you'd like to run some tests to compare, there's a simple way to do it. Numpy's `Chebyshev` class returns a callable `Polynomial` object $p_N$ that you initialize with an array of Chebyshev coefficients. It evaluates $p_N$ using Clenshaw recurrence. Now that you can generate Chebyshev coefficients quickly, you can create a `Chebyshev` instance and a barycentric interpolator (from Problem 1) for the same underlying set of $x_j, f_j$ and compare their evaluation speed and accuracy. You don't have to submit this — just making you aware of it in case you want to explore.

Note that Clenshaw recurrence is not limited to Chebyshev representations of functions. You can use the process to evaluate any $f(x)$ whose expansion coefficients you know in a set of basis functions that satisfy a 2nd-order recurrence relation of a given form (which a lot of commonly used basis sets do — see *Numerical Recipes*). But when $f$ is specifically an interpolating polynomial to

some other function, I tend to prefer the barycentric formula, mostly because it doesn't require you to find any expansion coefficients in advance. Still, if you're computing Chebyshev coefficients of an interpolating polynomial anyway for some other reason, then I think it's a toss up whether to evaluate that polynomial barycentrically vs. Clenshaw-ically.