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 jth Lagrange basis polynomial evaluates the difference between x and every node *except* the jth 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

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

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

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

(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 $x_0, x_1, \ldots, x_N$ 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 (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 jth 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 kth 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.