Finite Differences and Quadrature

Varun Shankar

January 21, 2019

1 Introduction

In this document, we discuss numerical integration (aka quadrature) and finite differences (FD). As promised, we will view these techniques from the perspective of polynomial interpolation, and also using Taylor series for the latter. While both techniques naturally admit an error analysis, we will argue that polynomial interpolation is the more powerful technique of the two in the following ways:

1. FD and quadrature formulae for unequally-spaced data points are easily generated from polynomial interpolation.
2. Error estimates for unequally-spaced FD and quadrature formulae are also easily obtained with polynomial interpolation.
3. Polynomial interpolation makes it easy to obtain formulae for *other* linear operators than just derivatives and integrals!

2 Finite Differences

Finite differences are numerical approximations to derivatives of a function using a finite number of samples of the function. FD formulae can be obtained using a variety of approaches, all equivalent. We will now review the most important approaches.

2.1 FD from the limit definition of a derivative

This is how most people are introduced to finite differences. The idea is to take the limit definition of a derivative, and drop the limit. For example, consider the following definition of the first derivative of a function

$$\frac{\partial f}{\partial x} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{1}$$

Dropping the limit symbol and assuming that h is sufficiently small, we get

$$\frac{\partial f}{\partial x} \approx \frac{f(x+h)-f(x)}{h}. \tag{2}$$

Using this approach, it is possible to approximate derivatives of arbitrary order. Unfortunately, there are a couple of drawbacks: a) this is a *symbolic* method of approximating the derivative (requiring some kind of symbolic math package to be able to do effectively) and b) this method does not intuitively give us an estimate of the *error* in the approximation to the derivative.

Regardless, we can use this approach as an opportunity to define a few terms. As in the case of polynomials, let $y_k = f(x_k), k = 0 \dots N$. Then, we define the *forward difference* as

$$\frac{\partial f}{\partial x} \approx \frac{y_{k+1}-y_k}{h}, \tag{3}$$

the *backward difference* as

$$\frac{\partial f}{\partial x} \approx \frac{y_k - y_{k-1}}{h}, \tag{4}$$

and the *central difference* as

$$\frac{\partial f}{\partial x} \approx \frac{y_{k+1}-y_{k-1}}{2h}. \tag{5}$$

We will rarely use the limit definition of the derivative in this class. This definition fails to show you the deep connections between finite differences and function approximation.

2.2 FD from Taylor series

The most common approach to deriving finite difference formulas is using Taylor series expansions. For example, letting $y_k = f(x_k)$ and $y_{k+1} = f(x_k + h)$, where h is the (equal) spacing between the nodes $x_k, k = 0 \dots N$, we can write the Taylor series for y_{k+1} as

$$y_{k+1} = f(x_k + h) = \underbrace{f(x_k)}_{y_k} + h\, f'(x)\big|_{x=x_k} + \frac{h^2}{2!}\, f''(x)\big|_{x=x_k} + \frac{h^3}{3!}\, f'''(x)\big|_{x=x_k} + \dots. \tag{6}$$

There are an infinite number of terms in this series. Rearranging a little, we can write

$$h\, f'(x)\big|_{x=x_k} = y_{k+1} - y_k - \frac{h^2}{2!}\, f''(x)\big|_{x=x_k} - \frac{h^3}{3!}\, f'''(x)\big|_{x=x_k} + \dots. \tag{7}$$

Dividing through by h and truncating the infinite series, we can write

$$f'(x)|_{x=x_k} = \frac{y_{k+1} - y_k}{h} + O(h), \tag{8}$$

where the symbol $O(h)$ simply says that the leading order term of the series (the largest term) is a term proportional to h. This is true because h is typically a small number less than 1. The $O(h)$ term is therefore the "error" in approximating the first derivative with the forward difference $\frac{y_{k+1}-y_k}{h}$. This approach is more systematic and useful than the limit approach of defining an FD formula; we got an error estimate for free! It is also possible to find an FD backward difference formula by expanding y_{k-1} and rearranging terms.

How do we compute an FD formula for the second derivative? We will need the Taylor series for both y_{k+1} and y_{k-1}.

$$y_{k+1} = y_k + h\, f'(x)|_{x=x_k} + \frac{h^2}{2!}\, f''(x)|_{x=x_k} + \frac{h^3}{3!}\, f'''(x)|_{x=x_k} + \ldots, \tag{9}$$

$$y_{k-1} = y_k - h\, f'(x)|_{x=x_k} + \frac{h^2}{2!}\, f''(x)|_{x=x_k} - \frac{h^3}{3!}\, f'''(x)|_{x=x_k} + \ldots. \tag{10}$$

Adding these two formulas, re-arranging and truncating, we get

$$f''(x)|_{x=x_k} = \frac{y_{k+1} - 2y_k + y_{k-1}}{h^2} + O(h^2). \tag{11}$$

This is a *centered* difference approximation to the second derivative. Much like the forward and backward difference approximations to the first derivative, it is also possible (but tedious) to obtain *one-sided* approximations to the second (and higher derivatives).

The above exposition should make clear the main weakness of the Taylor series approach: it is tedious! Imagine generating an FD formula for the fourth derivative; you would need four Taylor expansions, and would have to combine them in the correct way to get a nice formula that cancels out the first through third derivatives. In the oil industry, they use 20^{th} order FD formulas for simulations; imagine doing that with a Taylor expansion!

Also, note that we assumed that the node spacing h is the same throughout! What if the nodes are unevenly-spaced? How would you generalize the above Taylor expansions? What would the error estimates look like? Clearly, the Taylor approach becomes cumbersome in realistic scenarios.

2.3 FD from polynomial interpolation

A very general and powerful technique for generating FD formulae in 1D (and on certain grids in higher dimensions) is via polynomial interpolation. The idea is to interpolate the function with a polynomial interpolant, then differentiate the interpolant. Since we already have an error estimate for polynomial interpolation, we can generate an estimate for derivatives of the interpolant as well.

Recall the error associated with polynomial interpolation of a function $f \in C^{N+1}[a,b]$:

$$f(x) - p(x) = \frac{f^{(N+1)}(\xi_x)}{(N+1)!} \prod_{k=0}^{N} (x - x_k), \tag{12}$$

where $\xi_x \in [a,b]$. Let $e(x) = \prod_{k=0}^{N} (x - x_k)$. Differentiating the interpolant, we get

$$f'(x) = p'(x) + \frac{f^{(N+1)}(\xi_x)}{(N+1)!} e'(x) + \frac{e(x)}{(N+1)!} \frac{d}{dx} f^{(N+1)}(\xi_x). \tag{13}$$

Recall that this formula is true for points x that are *not the interpolation nodes.* If $x = x_j$ is one of the interpolation nodes, $e(x)$ is zero. This throws away the last term in the above expression, giving us

$$f'(x_j) = p'(x_j) + \frac{f^{(N+1)}(\xi_{x_j})}{(N+1)!} e'(x_j). \tag{14}$$

We therefore have error estimates for the derivative of the polynomial interpolant at the interpolation nodes *and* at nodes not within the set of interpolation nodes! For derivatives of order n, at the interpolation nodes, we have

$$f^{(n)}(x_j) = p^{(n)}(x_j) + \frac{f^{(N+1)}(\xi_{x_j})}{(N+1)!} e^{(n)}(x_j). \tag{15}$$

Unlike Taylor series expansions, we get a very general error estimate that does not depend on equispaced nodes of spacing h. However, we should verify that the Taylor series method and the polynomial interpolation method yield the same formula.

An Example

Consider the case of $N = 1$ and $j = 0$, with $n = 1$. In other words, we have two interpolation nodes x_0 and x_1, and we require an estimate to $f'(x_0)$. The Lagrange interpolating polynomial is

$$p(x) = y_0 \ell_0(x) + y_1 \ell_1(x), \tag{16}$$

with

$$\ell_0(x) = \frac{x - x_1}{x_0 - x_1}, \tag{17}$$

$$\ell_1 = \frac{x - x_0}{x_1 - x_0}. \tag{18}$$

We know that $p'(x) = y_0 l_0'(x) + y_1 l_1'(x)$. The derivatives of the basis functions are easily computed as

$$\ell_0'(x) = \frac{1}{x_0 - x_1}, \tag{19}$$

$$\ell_1'(x) = \frac{1}{x_1 - x_0}, \tag{20}$$

thus giving us

$$p'(x_0) = \frac{y_0}{x_0 - x_1} + \frac{y_1}{x_1 - x_0}, \tag{21}$$

$$\implies f'(x_0) = \frac{y_0}{x_0 - x_1} + \frac{y_1}{x_1 - x_0} + \frac{1}{2} f''(\xi)(x_0 - x_1), \tag{22}$$

$$\implies f'(x_0) = \frac{y_1 - y_0}{x_1 - x_0} - \frac{1}{2} f''(\xi)(x_1 - x_0). \tag{23}$$

Notice that we have made *no assumptions* about node spacing here, but have nevertheless been able to derive an FD formula for the first derivative of the function f at the nodes; even more importantly, we have an error estimate involving the second derivative of the function and the spacing between the nodes! Of course, if $h = x_1 - x_0$, we have

$$f'(x_0) = \frac{y_1 - y_0}{h} + O(h), \tag{24}$$

which is the *forward difference* approximation to the first derivative that we saw previously. We have demonstrated that the three approaches discussed thus far are equivalent.

Observe what just happened. We fit an interpolant through two points. This is clearly a line, so the process was linear interpolation. Linear interpolation has an error that grows as $O(h^2)$. However, when we estimated the first derivative, the error was $O(h)$. Thus, for each derivative of the polynomial interpolant, we lose one order of convergence.

This has ramifications; for example, if you are attempting to compute a second derivative $n = 2$, you cannot use an interpolant with $N = 1$. Why? Well, if you differentiate that linear interpolant twice, the error is $O(1)$; in other words, decreasing the point spacing will not reduce the error in any way! This will be a non-convergent FD formula. In general, this implies that you want $N \geq n$, where N is the degree of the polynomial interpolant and n is the order of the derivative.

Note also that it would be unwise to use a very high-degree polynomial interpolant to approximate a function that is only finitely-smooth. After all, the error term depends solely on the derivative of the function! In general, we must try to predict the required order of the polynomial interpolant to our data based on our knowledge of the source of the data. These are real-world problems with generating FD formulae.

NOTE 1: In case you missed it in our discussion thus far, the FD coefficients are simply derivatives of the Lagrange basis $\ell(x)$. In practice, for large N, you would use the barycentric form of the Lagrange basis, and differentiate that instead. This is one way to generate those 20^{th} order FD formulae we mentioned earlier!

NOTE 2: If you have trouble differentiating the function $e(x)$ and doing an error analysis for derivatives of polynomial interpolation at equispaced points, don't worry! You can simply do a Taylor expansion of the y terms other than at the point where we wanted the derivative; in this case x_0 is that point, so we would Taylor expand $y_1 = y_0 + h$. This will automatically yield the error estimate for the FD formula.

2.4 Equivalent approaches

Even within the polynomial interpolation framework, there are 3 major approaches to generating the FD weights of interest, and they are all equivalent.

1. For small N, form the Vandermonde system V, solve for the coefficients a_k, store them, then multiply by nx_j^{n-1} to obtain the n^{th} derivative of the function at the point x_j.
2. Alternatively, for larger equispaced N, differentiate the Lagrange basis, evaluate the derivative of the basis functions at the point x_j and multiply these derivatives with their associated y_k values.
3. Let the unknown FD weights be $w_k, k = 0 \dots N$. Then, the weights can be directly computed by solving the following system:

$$\underbrace{\begin{bmatrix} 1 & x_0 & x_0^2 & x_0^3 & \dots & x_0^N \\ 1 & x_1 & x_1^2 & x_1^3 & \dots & x_1^N \\ 1 & x_2 & x_2^2 & x_2^3 & \dots & x_2^N \\ \vdots & \vdots & \vdots & \ddots & \vdots & \\ 1 & x_N & x_N^2 & x_N^3 & \dots & x_N^N \end{bmatrix}}_{V} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} = \begin{bmatrix} \frac{\partial^n}{\partial x^n} 1\big|_{x=x_j} \\ \frac{\partial^n}{\partial x^n} x\big|_{x=x_j} \\ \frac{\partial^n}{\partial x^n} x^2\big|_{x=x_j} \\ \vdots \\ \frac{\partial^n}{\partial x^n} x^N\big|_{x=x_j} \end{bmatrix}. \tag{25}$$

This last one is surprising, but can be proved to be equivalent to the first two quite straightforwardly.

2.5 Other linear operators

One of the advantages of the polynomial interpolation approach is that we could use it to generate FD weights for other linear operators as well; again, we will be able to obtain two error estimates for free: one at the nodes x_k and one for nodes in between the x_k values.

Let $\mathcal{L}$ be the linear operator. Then, at the nodes, we have

$$\mathcal{L}f(x_j) = \mathcal{L}p(x_j) + \frac{f^{(N+1)}(\xi_{x_j})}{(N+1)!}\mathcal{L}e(x_j), \tag{26}$$

while the more general estimate for errors at non-nodal evaluation points is

$$\mathcal{L}f(x) = \mathcal{L}p(x) + \mathcal{L}\left[\frac{f^{(N+1)}(\xi_x)}{(N+1)!}e(x)\right]. \tag{27}$$

2.6 Other approaches

Even within the polynomial framework, there are robust and faster algorithms for computing FD coefficients. See “Calculations of Weights in Finite Difference Formulas”, Bengt Fornberg, 1998, for two such algorithms.

Note that while we used polynomials above, this may not be possible. For example, if your grid is equispaced, polynomial interpolants of high-degree are infeasible due to the Runge phenomenon. Alternatively, you may know in advance that the function you are differentiating is periodic, and that you can do better than polynomial interpolation. The following alternatives are sometimes used:

1. Chebyshev FD: build a polynomial interpolant on Chebyshev nodes or extrema translated to the interval of interest, and differentiate that interpolant.
2. Spline FD: build FD coefficients by differentiating spline interpolants, which are low-degree piecewise polynomial interpolants built on equispaced points and “stitched” together to obtain global continuity, smoothness and convergence.

The other issue is that polynomial interpolation can easily fail for scattered node configurations in 2D and higher (the Vandermonde matrices can be singular). Unless the nodes lie on special grids (Cartesian grids, triangles, tetrahedra, etc.), the polynomial approach to generating FD coefficients fails!

A new approach to generating FD formulae in arbitrary dimensions is by differentiating Radial Basis Function (RBF) interpolants; RBFs have been shown to be generalizations of polynomials *and* Fourier series, and more importantly, have been shown to be non-singular for distinct (scattered) interpolation nodes in arbitrary dimensions. Even better, when used in 1D, they will recover the polynomial FD weights in some limit. We will discuss RBFs in greater detail later in the semester.

3 Quadrature

Quadrature formulae are numerical approximations to definite integrals of a function using a finite number of samples of the function. While quadrature formulae, like FD formulae, can be obtained with a variety of approaches as well, we will restrict our attention to discussing how quadrature can be generated using polynomial interpolation in the Lagrange form.

3.1 Quadrature from polynomial interpolation

The key idea is that to approximate an integral, we will integrate the Lagrange basis, much like we differentiated the Lagrange basis for the FD formula. To obtain error estimates for quadrature rules, we will simply integrate the polynomial error estimate. In other words, we have

$$\int_a^b f(x)dx = \int_a^b p(x)dx + \int_a^b \left[\frac{f^{(N+1)}(\xi_x)}{(N+1)!}e(x)\right]dx. \tag{28}$$

This is a bit cumbersome to work with. To make this simpler, we note that if $f^{(N+1)}$ has a maximum, we can set

$$M_{N+1} = \max_{x\in[a,b]} |f^{(N+1)}(x)| < \infty. \tag{29}$$

This transforms the above equality to an upper bound, but gives us

$$\left|\int_a^b f(x)dx - \int_a^b p(x)dx\right| \leq \int_a^b \left[\frac{M_{N+1}}{(N+1)!}e(x)\right]dx. \tag{30}$$

We can now yank out the constants from the integral, giving us

$$\left|\int_a^b f(x)dx - \int_a^b p(x)dx\right| \leq \frac{M_{N+1}}{(N+1)!}\int_a^b e(x)dx. \tag{31}$$

Example

Consider the Trapezoidal Rule, a closed Newton-Cotes type formula (a fancy way of saying the points are equally-spaced, and that we are using the end-points). Let $[x_0, x_1]$ be the interval under consideration. Then, we have

$$\int_{x_0}^{x_1} f(x)dx \approx y_0 \int_{x_0}^{x_1} \ell_0(x)dx + y_1 \int_{x_0}^{x_1} \ell_1(x)dx, \tag{32}$$

$$\ell_0 = \frac{x - x_1}{x_0 - x_1}, \tag{33}$$

$$\ell_1 = \frac{x - x_0}{x_1 - x_0}. \tag{34}$$

Simplifying, we get

$$\int_{x_0}^{x_1} f(x)dx \approx \frac{1}{2}(x_1 - x_0)(y_0 + y_1). \tag{35}$$

It should be clear that $N = 1$ here; this is an integral of a linear interpolant. Now, using the previously developed error bound, we have

$$\left| \int_{x_0}^{x_1} f(x)dx - \int_{x_0}^{x_1} p(x)dx \right| \leq \frac{M_2}{2!} \int_{x_0}^{x_1} e(x)dx, \tag{36}$$

$$\leq \frac{M_2}{2} \int_{x_0}^{x_1} (x - x_0)(x - x_1)dx, \quad \leq \frac{M_2}{12}(x_1 - x_0)^3. \tag{37}$$

If $h = x_1 - x_0$, then the trapezoidal method yields $O(h^3)$ convergence in a single interval. In practice, it is common to divide up an interval into subintervals of width h_x. The individual $O(h_x^3)$ errors then add up to $O(h_x^2)$ asymptotically.

Note 1: The exact opposite of differentiation happened here with the error; integrating the polynomial gave us an extra order of convergence. To get higher orders, use higher degree polynomials.

Note 2: Integrals are quantities that live over entire intervals. As such, in addition to the degree of the polynomial interpolant, we have another important choice to make: where to sample the function in the interval. Often, this is not under our control, but when it is, it is typical to select nodes with certain properties (say, clustering towards the ends of the interval like in Chebyshev nodes or extrema). It is not uncommon to *design* a numerical method solely around a predetermined (wise) choice of quadrature points (nodes).

Note 3: For spatial integration, Gaussian quadrature (based on Legendre zeros) and Clenshaw-Curtis quadrature (based on Chebyshev zeros or extrema) are unbeatable, with the former being faster and more accurate for a larger class of functions than the latter. Note that this wasn't always the case. There was a time (until 2012) when, for a large class of functions, CC quadrature would give the same accuracy as Gaussian quadrature for a lower computational cost.